A pragmatic analysis of scheduling environments on new computing platforms

Lionel Eyraud

Laboratoire ID-IMAG, Institut National Polytechnique de Grenoble,
51 avenue Jean Kuntzmann, 38330 Montbonnot Saint-Martin, France

Abstract

Today, large scale parallel systems are available at relatively low cost. Many powerful such systems have been installed all over the world and the number of users is always increasing. The use of such clusters require special administration tools, which have been developed recently, and most frequently rely on intuitive heuristics to solve the underlying difficult scheduling problems. On the other hand, recent theoretical work has designed a number of models, such as the Parallel Task model, specifically aimed at cluster environments.

The objective of this work is to study and analyze the path from theoretical models and algorithms to their implementation on an actual environment on a real platform. We outline in details the divergences between classical models and a real batch scheduling system, and propose solutions to adapt a theoretically well-founded algorithm to such a system. Experimental results show that this implementation perform about as well as FCFS with backfilling, and much better in the difficult instances. We hope that further usage of this algorithm in a live system will show that interaction between theory and practice can be fruitful.

1 Introduction

1.1 Scheduling in clusters

The last few years have been characterized by huge technological changes in the area of parallel and distributed computing. Today, powerful machines are available at low price everywhere in the world. The main visible line of such changes is the large spreading of *clusters* which consist in a collection of tens or hundreds of standard almost identical processors connected together by a high speed interconnection network [6]. The next natural step is the extension to local sets of clusters or to geographically distant grids [14].

In the last issue of the Top500 ranking (from November 2005 [3]), 360 entries are clusters sold either by IBM, HP or Dell. Looking at previous rankings we can see that this number (within the Top500) is still increasing rapidly. It is also interesting to remark that while cluster-based systems occupy 72% of the list, their total performance represents about 50% of the total performance.

This democratization of clusters calls for new practical administration tools. Even if more and more applications are running on such systems, there is no consensus towards an universal way of managing efficiently the computing resources. Current available scheduling algorithms dedicated to cluster environment were mainly created to provide schedules with performance guarantees for the makespan criterion (maximum execution time of the last job); however the corresponding theoretical models are always simplified with respect to the real setting of a cluster. Moreover, most of them are pseudo-polynomial, therefore the time needed to run these algorithms on real instances and the difficulty of their implementation is a drawback for a more popular use.

On the other hand, software solutions have been developed to be able to use such machines: these tools are called *batch schedulers*; classic examples are PBS/OpenPBS, LSF, NQS, etc. The review [4] lists these major systems and their features. But since the theoretical work on scheduling algorithms is not well adapted to a real implementation, and often not well-known in the software community, the algorithms implemented in such systems are always simple online heuristics, without any theoretical foundations.

1.2 Contribution of this work

The objective of this work is to study and analyze the path from theoretical models and algorithms to their implementation on an actual environment on a real platform.

This paper reports the adaptation and implementation of a scheduling algorithm, namely DEMT [10], into an actual batch scheduling system. This algorithm is based on a new scheduling method, inspired by several existing theoretically well-founded algorithms. It aims at optimizing two criteria simultaneously: the makespan $C_{\max}$, the completion time of the last job, which is a system-centric measure representing the utilization of the platform ; and the minsum $\sum C_i$, which can also be seen as the average completion time, and is a rather user-centric measure.

The makespan is historically the main objective function. However, this criterion is interesting only if we consider the tasks altogether and from the viewpoint of a single user. If the tasks have been submitted by several users, as is the case for grid computing, other criteria have to be considered. If we consider that each user is interested in minimizing the completion time of his or her jobs, selecting the minsum, or average completion time, seems to be a good choice. It is even possible to assign a weight to each job, and minimize the weighted sum of completion times. This allows to express a difference in priority between certain tasks. However, since this user-centric optimization should not degrade performance too much, it is important to be also concerned with the makespan criterion.

This implementation was performed in an existing batch scheduling middleware called OAR [5]. This batch scheduler, developed locally, is based upon an original design that emphasizes on low software complexity by using high

level tools. This system offers most of the important features implemented by other batch schedulers such as priority scheduling (with queues), reservations, backfilling, and some global computing support. It is is used in a large number of computing clusters in Grenoble, and is currently one of the basic blocks of the french national Grid project Grid'5000.

When using and interacting with an actual scheduling environment like OAR, some conceptual differences with the classical theoretical models become apparent. These differences are often large enough to require a very deep rethinking; it is beyond the scope of this paper to give a complete solution to these problems. Instead, we will try to expose the reasons for these discrepancies, and what can be done on a practical side to deal with an actual environment while keeping the main ideas of the theoretical results.

This paper is organized as follows. In section 2, we will present cluster scheduling from the viewpoint of the theoretical community, and the key ideas that are the basis for the DEMT algorithm. The key features of OAR will be presented in section 3, as well as the flaws (or limitations) of the classical theoretical models. Section 4 will be devoted to the practical solutions that can be used to overcome these limitations, and to experimental results using simulated runs of real traces from our clusters.

2 Theoretical Context

The target execution platform considered here is a cluster made of a medium number of nodes (from some dozens to several hundreds of SMP or PC machines) which are fully connected and (almost) homogeneous. Jobs are submitted via some specific entry nodes, and are affected to priority queues. Typically, a job represents a request to gain access to a fixed number of nodes for a fixed period of time.

The application model that is best suited to study scheduling in a cluster environment is the *Parallel Tasks* model. Informally, a Parallel Task (PT) is a *task* that regroups elementary operations, typically a numerical routine or a nested loop, and which contains itself enough parallelism to be executed by more than one processor. The classification of Feitelson et al. [13] distinguishes between:

- *rigid jobs*, for which the number of processors required is fixed by the user at submission time, and
- *moldable jobs*, for which the number of processors is determined by the scheduling algorithm before execution time. Each job can thus be given any number of nodes, between 1 and m (the total size of the cluster). In that case, the processing time of the task depends on the number of processors alloted to the task.

In any case, the number of processors does not change until the completion of the job. See [8] for a survey of results about the Parallel Task model.

For historical reasons, most submitted jobs are rigid. However, intrinsically, most parallel jobs are moldable. Indeed, most programming environments do not assume *a priori* the number of available processors; it is often specified at execution time. This is also true for many numerical parallel libraries. The main restriction, and admittedly the greatest limit of the moldable model, is that some memory-limited jobs cannot be executed on less than a given number of nodes, and the classical moldable model does not take these situations into account[1]. On a theoretical level, the moldable job model gives much more flexibility to the scheduler, allowing for more efficient algorithms and a better overall utilization of the cluster. For example, the best known guarantee for off-line scheduling of independent moldable tasks is 3/2 [19], whereas the best guarantee for the equivalent rigid problem is 2 [15]. For a detailed analysis of Parallel Tasks, see [11].

2.1 Key ingredients for guaranteed heuristics

The algorithm we have studied is DEMT, described in detail in [10] and [12]. It was designed for a model with moldable tasks, in an off-line setting, with the goal of minimizing both the weighted minsum and makespan criteria. The theoretical results make the additional classic assumption of *monotony* for the execution times of a moldable task. A moldable task is monotonic if its execution time does not increase with the number of processors, while its total work[2] does not decrease. This is based on the observation that a parallel task is more efficient when executed on fewer processors, because the overhead of parallelism is lower. Once again, this does not take into account memory constraints that would cause this application to revert to swapping, and thus take a much longer time to execute.

In order to have good results with respects to the minsum criteria, a common heuristic approach is to try and schedule the smallest tasks first. The DEMT algorithm is thus built on a quite classical structure of batches of increasing lengths, so that the smaller tasks, that can fit into the first smaller batches, are scheduled first. This batch structure has been studied in a number of algorithms presented in the literature.

The principle of the algorithm is presented in figure 2.1: the length of the batches are decided at the beginning of the scheduling, each batch being twice as long as the previous one. Then, for each batch, there are three consecutive steps:

- In the first step, we start by selecting the tasks that are not too long to run in the batch.
- In the second step, since the tasks are supposed to be moldable, we can select for each task the minimum number of processors that allows it to

[1] This is the reason why some moldable models assume that each job has a list of *possible allocations*, and are thus not able to to run on any number of processors

[2] The total work of a task is defined as the product of its execution time and the number of processors it is executed on.

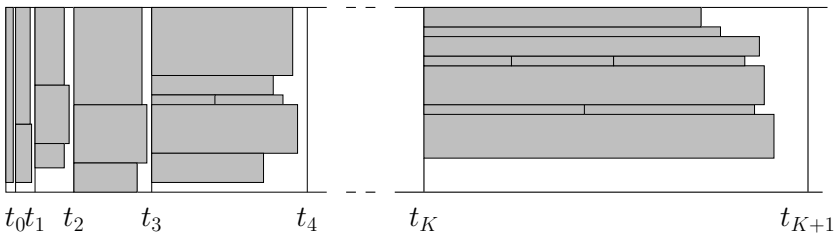

Figure 1: Principle of the DEMT algorithm

run in the batch.

- In the third step, a selection procedure chooses among these tasks, so as to maximize the total weight of the selected tasks.

These three steps are then repeated until there is no task left.

The interest of moldability in step 2 is that a relatively short task that couldn't be scheduled in the first shelves can be allocated less processors (increasing its processing time at the same time). This has the effect of lowering the waste of resource that results from the shelf structure: almost all tasks will have a processing time close to the length of the shelf.

The DEMT algorithm has two variants: the first variant is based on a structure of batches, using the clever 3/2-approximation algorithm of [19] to schedule the tasks inside each batch. In the off-line case, this variant was proved to have a dual approximation ratio of (3, 6) for the makespan, and the minsum criterion respectively. In an on-line case, a randomized version of this variant has an average performance guarantee of (3.88, 4.08) [9], which is much better than the previously best known guarantee of (8, 8.53). The second variant is simpler: it is based on a structure of shelves (a special case of batches, where all tasks start their execution at the same time), and the selection procedure consists of a knapsack algorithm. This variant has no known approximation ratio, but its behavior has been extensively tested on a set of generated instances simulating real jobs [10].

After all tasks have been assigned to a shelf, the first schedule is simple: we start all the selected tasks of one shelf at the same time. A straightforward improvement is to start a task at an earlier time if all the processors it uses are idle. A further improvement is to use a list algorithm with the shelf ordering and a local ordering within the shelves, as it allows to change the set of processors alloted to the tasks.

The desirable property of this algorithm, and what enabled its implementation despite the limitations of the model, is that the *scheduling* problem (choosing the start times of the tasks) and the *allocation* problem (choosing the nodes on which each task will be scheduled) are decoupled. During the selection phase, the time dimension does not interfere: all tasks that are too long to be scheduled

in the current shelf have been filtered out, and then the processing times of the tasks are not considered. This is why the shelf structure is what we are going to keep through all the transformations needed to adapt the algorithm to the actual OAR environment.

3 Practical Context

3.1 A batch scheduler – OAR

OAR [5] is a batch scheduling system, developed at our lab, which is based upon an original design that emphasizes on low software complexity by using high level tools. This system offers most of the important features implemented by other batch schedulers such as priority scheduling (with queues), reservations, backfilling, and some global computing support.

This software is used in a large number of computing clusters in Grenoble, and is also used as a basic block of the Grid'5000 French Grid project. It is a full production system, and has been designed with enough modularity in mind to allow relatively easy development of alternative scheduling policies.

OAR has the following features:

- OAR handles submission and deletion of jobs, with priority queues.
- OAR supports clusters of SMPs: with each node is associated a "maximum weight", and each job has a "weight". A node can only run tasks whose weights don't add up to more than its maximum weight. This notion of weight allows a non-exclusive access to a given node, in order to take advantage of several processing units on the same node.
- OAR handles *properties*: users can select which nodes are candidates for running their job. This feature is important because in a real system, nodes are not always exactly homogeneous: some might have more memory than others, or some local hard disks might be more prone to failures; the connectivity is also an important issue (nodes that are physically plugged to the same switch communicates faster than nodes with a longer communication route).
- OAR handles reservations: it is possible for a user to ask for nodes ahead of time. If the reservation is accepted, the system guarantees that it will start on time. The reservation system is useful in at least two cases: for Grid Computing, when users want to run their application at several different distant sites, reservation is a way to make sure that the application starts at the same time on all sites; for demonstration purposes, when a user wants to show the operation of an application on a scheduled meeting.
- OAR does not handle moldable tasks: the job model is a rigid one. Even though most jobs are intrinsically moldable (because most programming environments do not assume *a priori* the number of available processors),

filling the moldable model with the values indicating how long the job will last for every possible allocation is a tedious work, and very few users are willing to spend the time necessary for it.

It is interesting to remark that these features are not specific to OAR: most job schedulers implement them too, because they are really needed by the users of the computing centers.

Scheduling in OAR The scheduling algorithm natively implemented in OAR is First Come First Served (FCFS), with conservative backfilling. Such an algorithm considers all tasks in the order of their arrival in the system, and greedily schedules each task at the earliest possible date, without delaying any previously scheduled task. It might happen that a given task x gets to run before another task y that was submitted before it, but in that case the task y could not have been scheduled earlier, even if x was not present.

This kind of algorithms are common in the batch scheduling literature [4], with several more aggressive variants that allow a task to delay an earlier task if it can be scheduled right now. Aggressivity improves the utilization of the machine, but it may make it possible for a job to *starve*, being constantly delayed by other smaller jobs arriving in a continuous stream.

From a theoretical point of view, FCFS, even with conservative backfilling, has no constant performance guarantee for the makespan criterion. Indeed, on a machine with m nodes, it is possible to build an instance with optimal makespan 1, and whose resulting FCFS schedule has makespan m.

3.2 Limitations of the models

On-line setting As all batch scheduling systems, the environment of OAR is a highly on-line one: all the events that take place in the future of the current scheduling time are completely unknown to the scheduler. This is especially true for the submission of tasks, but also for the completion of currently running tasks. Indeed, it is important to note that the semantics of the submission system means that jobs require an access to a certain number of nodes for a certain period of time (this is called the *walltime* of the job), but do not specify even an estimation of their real running time. When a job runs for longer than its walltime, it is killed by the system to allow other waiting jobs to be executed. The walltime of a job is thus an upper bound of its actual running time; and it is often a very large one, since users do not want to see their jobs killed because of a bad estimation[3]. This is actually in conflict with classical clairvoyant models, which assume that the execution times of the jobs are precisely known; on the other hand, non-clairvoyant models assume a complete lack of knowledge about the running time of the jobs. Some recent theoretical studies about the *robustness* of scheduling algorithms against modifications of the input instance are interesting in this context, but they often assume that

[3]With this in mind, it would be interesting to study the effect of a scheduling algorithm biased towards jobs with smaller walltimes on the submission habits of the users

these modifications are not too large. We can see here that the environment of an actual batch scheduling system is somewhere in the middle between the two extreme clairvoyant and non-clairvoyant models.

In OAR, this on-line setting is handled in a conservative way: at a given time, all decisions are taken as if the available information was exact. When an external event occur (a submission of a new job or the completion of a job before its walltime), the scheduling process is restarted with the new state of the system; of course, already running jobs are not re-scheduled. The result of this behavior is that all scheduling decisions about the future are only predictions; they are accessible to the user for information, but they are subject to change with the state of the system.

Another interesting discussion concerning on-line scheduling is the question of optimization criteria. Actually, when scheduling in an on-line context, the completion metrics like the makespan or average completion time make less sense. Indeed, their usage intrinsically assumes that there is a "time zero" in time against which all completion times can be compared. Let us imagine, in a schedule, two tasks being executed one at the beginning of the schedule and the other near the end. If both tasks are delayed by one hour, the sum of the completion time will be increased by two hours, but the relative increase will be much higher for the first task than for the second. But from the point of view of the second task, the important quantity is how much time it has waited in the queue: if one additional hour of waiting doubles this time for both tasks, then the second task has been handicapped by the same proportion than the first.

This is why it would have more sense to study *flow* metrics. The flow time of a job is defined as the difference between its completion time and its release time; it represents the time that the job spent in the system. Of course, when one is concerned with finding an optimal schedule, the sum of completion times and the sum of the flow times are equivalent; but the difference arises when trying to design approximation algorithms. Some theoretical studies [18] have looked at the max-flow criterion, as well as the (weighted) average flow. But these criteria appear much more challenging to handle, and they haven't been addressed in the context of parallel tasks yet.

Scheduling with reservations With the increasing interest in large scale computing systems, in which several clusters are connected and interoperable, more and more scheduling systems feature advanced reservations. On the theoretical side, the problem of scheduling in presence of machine unavailability is rather difficult: even very simple problems become NP-Hard when this additional constraints is considered (like for example the makespan minimization on a single machine). For this reason, machine unavailability has been studied mostly in quite simple models, with either preemption or on one single machine, and never in the parallel tasks model. A quick survey of these studies can be found in [16].

A note about heterogeneity The support of properties dramatically modifies the platform model: in OAR, like in most cluster-oriented batch schedulers, it is possible to distinguish between two computing nodes.

From the user's point of view, this differentiation of the nodes allows to express a boolean predicate to specify which subset of the nodes are eligible to run his or her job. This is often desirable because of physical differences between nodes (available memory, local hard disk speed, networking card). Another issue is the connectivity between nodes: when a cluster is reasonably large, it is impossible to physically connect all of the nodes on a single switch. Then network latency between nodes connected on two different switches is slightly higher (this is the limit of the "fully connected" machines model); this difference is negligible for most applications, but some network intensive applications (especially benchmarking ones) need to be scheduled on nodes that share the same switch.

This type of heterogeneity is very far from classical theoretical models, in which nodes can be either *homogeneous*, *related* or *unrelated* [17]. In all these cases, it is only the processing speed of the machines that is taken into account[4]. Of course, it would be possible to use the unrelated model to take this feature into account, assigning a very high computation time for an impossible (task, node) pairing. But the unrelated model is too general and too complex to be able to design efficient algorithms in such a setting; it is actually unclear how to define the parallel tasks model in an unrelated machine model. Once again, the platform model of actual cluster environments is somewhere in the middle between the two extreme homogeneous and unrelated models.

4 From theory to reality

This section will describe the concrete choices that had to be made to adapt the DEMT algorithm to run in the OAR environment. We hope that this will show that it is actually possible to implement a clever algorithm in a real environment, despite all the differences with the original models.

4.1 Adaptation to on-line setting

The DEMT algorithm was designed for both off-line and on-line settings. However, the very structure of the algorithm, with shelves of increasing sizes, assumes that there is an "origin", a point in time where everything starts. It would be impracticable to use the algorithm as it is in a real environment, since the shelves sizes would rapidly grow very large, without any connection with the length of the tasks. On the other hand, it would not be possible either to restart the algorithm with small shelves each time a task is submitted, since this

[4]Interestingly, nodes in batch scheduling systems are assumed to be homogeneous in terms of computing speeds, or at least are not explicitly heterogeneous: the walltime, which is the closest approximation to the processing time of a job, is given independently of the nodes it will be scheduled on.

would be too highly biased towards small tasks and lead to possible starvation of larger tasks.

In order to keep the structure in shelves of the algorithm, we decided to take an intermediate approach, and to "reset" the shelves lengths periodically. Every day, at the beginning of the morning and of the afternoon (specifically at 8am and 1pm, but these choices could be changed), the DEMT algorithm is restarted with small shelves again. Between two restart times, the algorithm behaves like the original DEMT, and reacts to the submission of a task by inserting it in the shelf structure at the time it is submitted. The rationale of this choice was to favor small tasks during the day, when interactivity is a bigger concern, and schedule larger tasks, that can afford some delay, during the night.

4.2 Rigid tasks

The DEMT algorithm was designed for moldable tasks, with the idea that future batch scheduling systems would include a support for this different job model. But OAR, like all actual batch schedulers, does only support parallel rigid jobs. Actually, this is not a big drawback: the original batch structure framework was designed in a very generic way, and is adaptable to many different job models.

Similarly, the shelf structure of the second variant can also handle rigid tasks. The allocation step of the procedure is simply bypassed, since the allocation is fixed when dealing with rigid jobs. With this modification, the good property that tasks inside each shelf are about the same length is lost. Hopefully, the compaction phase will help improve the overall utilization, even without this property.

It is interesting to note that it will be easy to extend this algorithm to an environment with moldable jobs mixed with rigid jobs. Since it is planned that the next version of OAR will include a support for moldable tasks, this is an important property.

4.3 Handling reservations

Like most scheduling algorithms, the DEMT assumes that all of the computing nodes are available and can execute a task. In reality, because of advanced reservations, this assumption is not true.

During the implementation of DEMT, we decided to start with a naive approach to solve this problem. The main advantage of scheduling tasks on shelves is that during the selection process, the time dimension does not matter: during this phase, the goal is only to select tasks that can be scheduled all at the same time and totalize as much weight as possible. In order to keep this desirable property, we decided to consider only the nodes that are available throughout the whole duration of the current shelf. The selection process then takes place as if the other nodes were absent. This choice allows to be sure that the set of the selected tasks can be scheduled all at the same time in the shelf.

This limitation is only in place during the selection procedure. During the compaction phase, which uses a conservative backfilling algorithm to try and

schedule all tasks at an earlier date if possible, the reservations are considered with their original dates. However, it is important to remark that with this modification, the result of the selection depends on the starting date of a shelf. As a result, for the selection procedure to be as close to the reality as possible, we decided to perform the compaction phase just after the tasks of a given shelf were selected. In this way, the starting date of the next shelf can be set to the last completion time of the previous one, and the availability of the nodes for the next shelf is as close as possible to reality.

4.4 Non homogeneous platform

The last difficulty encountered when implementing the DEMT algorithm was complying to the heterogeneity model of OAR. In such a context, the selection problem turns out to be very complicated, even to formulate. Formally, it can be expressed in this way: the input is

- a set $\mathcal{N}$ of nodes, each node n_i having a number of available processors π_i,
- a set $\mathcal{T}$ of tasks, each task t_j being represented by a requested number of nodes q_j, a requested number of processors w_j, a set $\mathcal{P}_j$ of allowed nodes, and a weight ω_j.

The goal is to output a subset $\mathcal{S}$ of $\mathcal{T}$ and an allocation function $\sigma : \mathcal{S} \to \mathcal{P}(\mathcal{N})$ such that:

- each selected task is allocated its requested number of nodes from its allowed nodes:

$$\forall t_j \in \mathcal{S}, \sigma(t_j) \subseteq \mathcal{P}_j \text{ and } |\sigma(t_j)| = q_j$$

- The sum of the requested processors of all tasks scheduled to a given node does not exceed this node's number of available processors:

$$\forall n_i \in \mathcal{N}, \sum_{j \in B_i} w_j \leq \pi_i \text{ where } B_i = \{j \in \mathcal{S} \,|\, n_i \in \sigma(t_j)\}$$

The objective of this problem is to maximize the total weight of the selected tasks, $\omega(\mathcal{S}) \equiv \sum_{t_j \in \mathcal{S}} \omega_j$.

This problem is a very difficult one: it does contain the knapsack problem when $\mathcal{P}_j = \mathcal{N}$ and $\pi_i = 1$, but it can also be seen as a SetPacking problem when all $\pi_i = 1$. The SetPacking problem is NP-Hard in the strong sense, unlike the knapsack problem.

Even though this selection problem is untractable in the general case, it is interesting to note that in practice, there is very little difference, if any, between the $\mathcal{P}_j$s. Indeed, most users don't express any constraints on their jobs, and those who do express the same constraints for all of their jobs. It is then possible to consider as undifferentiable all the nodes that always appear together in the

$\mathcal{P}_j$s, and to form *groups* of nodes that are undifferentiable. With this collapsing, the size of the search space can be greatly reduced in most of the cases, and it is possible to apply an integer dynamic algorithm to solve it exactly. Of course, the size of the state space, as well as the complexity of the algorithm, is exponential in the number of groups, as well as in the largest π_i.

Nevertheless, this implementation performs very well on most cases. Of course, special difficult cases are met in practice. In order to keep a reasonable execution time, the selection procedure automatically reverts to a greedy algorithm when the problem size is too large (too many groups) or when the dynamic algorithm takes more than a certain predefined timeout.

5 Experimental analysis

5.1 Description of the environment

OAR is one of the basic blocks of the Grid'5000 French national project [1], and is thus used on each cluster to schedule all the local jobs. Although Grid'5000 is being extensively used by a great part of the French parallel computing community, there is little competition for the resources on most clusters, and there is not much interest in replaying the traces of these clusters. The only exception is the Icluster2 [2], consisting of 104 bi-Itanium 2 nodes, which has been operational much earlier than the other clusters, and is consequently used by a greater number of users, resulting in a number of large periods of high activity. We have thus replayed the trace of the Icluster2 for these high activity periods.

5.2 Simulations

For each one of these time windows, we have run an event-based simulator with a behavior exactly equivalent to that of OAR in the same situations. Because some real events, especially machine failures[5], cannot be simulated in this way, we have run the simulations with the two different schedulers: our DEMT-like implementation, and the original FCFS with conservative backfilling of OAR. The result of one simulation is a schedule that specifies the starting time and the machines allocated to each job. We can then compare, for each job, the difference between its starting times[6] under the two scheduling algorithms.

It is important to notice that the original traces were obtained in a system using the FCFS scheduling algorithm. With OAR, users have access to the state of the system, and even to the predictions of future scheduling decisions. It it thus very likely that the results will be biased towards the FCFS algorithm, because the jobs submitted are often adapted to fit in the holes of its schedule.

[5] Machine failures cannot be simulated exactly because they can only be detected by OAR when a job starts or finishes. With only the execution traces of the jobs, it is impossible to know when they would have been detected with a different schedule of the tasks

[6] Since the processing times are the same in both cases, the difference between the two completion times is the same as the difference between the two starting times.

Instance	Nb jobs	Nb identical jobs	Total diff.	Avg. diff.	Std dev.
2(a)	1391	362	113133	109.945	117.073
2(b)	1074	749	1824.79	5.61475	23.1384
2(c)	575	221	121.904	0.344362	10.6313
2(d)	429	364	-94.8261	-1.45886	25.825
2(e)	1292	757	-757.596	-1.41607	11.7721
2(f)	239	198	-93.0825	-2.2703	16.1927

Table 1: Numerical results of the simulation. Times are given in hours; a positive value means that the task was executed earlier with DEMT than with FCFS.

Table 1 summarizes the results of these simulations, and gives for each instance the total number of jobs and the number of *identical* jobs, i.e. the jobs whose starting time was the same with the two algorithms. These identical jobs are removed from the rest of the analysis, to put more emphasis on the distribution of the other jobs. Table 1 also shows the total difference of the starting times of all the tasks, with their average and standard deviation. Figure 2 show the histogram that represents the distribution of these differences of starting times, together with the average (the little line under the bars) and quartile values (the dotted lines). In both cases, any task with a positive value of the difference has been executed earlier with the DEMT algorithm than with the FCFS one.

The first observation is that a little more than half of the jobs are not affected by the change of scheduling algorithms. These are often the jobs which were submitted at times where the occupation of the machine was low, and could be executed right away. Another source of identical jobs are those which were submitted just before a reservation, and had to wait for the end of this reservation before being executed. Another interesting observation is that although the average difference is rather low (except for instance 2(a) which will be discussed below), the standard deviation is quite high. This can be observed on the graphs too, where the distribution is rather spread out, with some jobs having large differences in starting times. This happens to very long jobs, which last for several days, and whose walltime was of one or two weeks. These jobs sometimes can be scheduled right away, but delaying them slightly will cause them to have to be scheduled after a reservation that is weeks ahead, and so these jobs may suffer of a very large delay with one algorithm compared to the other.

On about half of the experiments, the average difference is slightly negative, around -1.5 hour. Interestingly, in these cases, the median value is a little higher than the average: this means that DEMT penalized some jobs by a large margin, but did not penalize so much the majority of the jobs. Nevertheless, on these experiments, the performance of DEMT is lower than that of FCFS.

Two experiments make a notable exception: figure 2(b), where the average difference is 5.61 hours, and figure 2(a), where it is 110 hours. This last value is

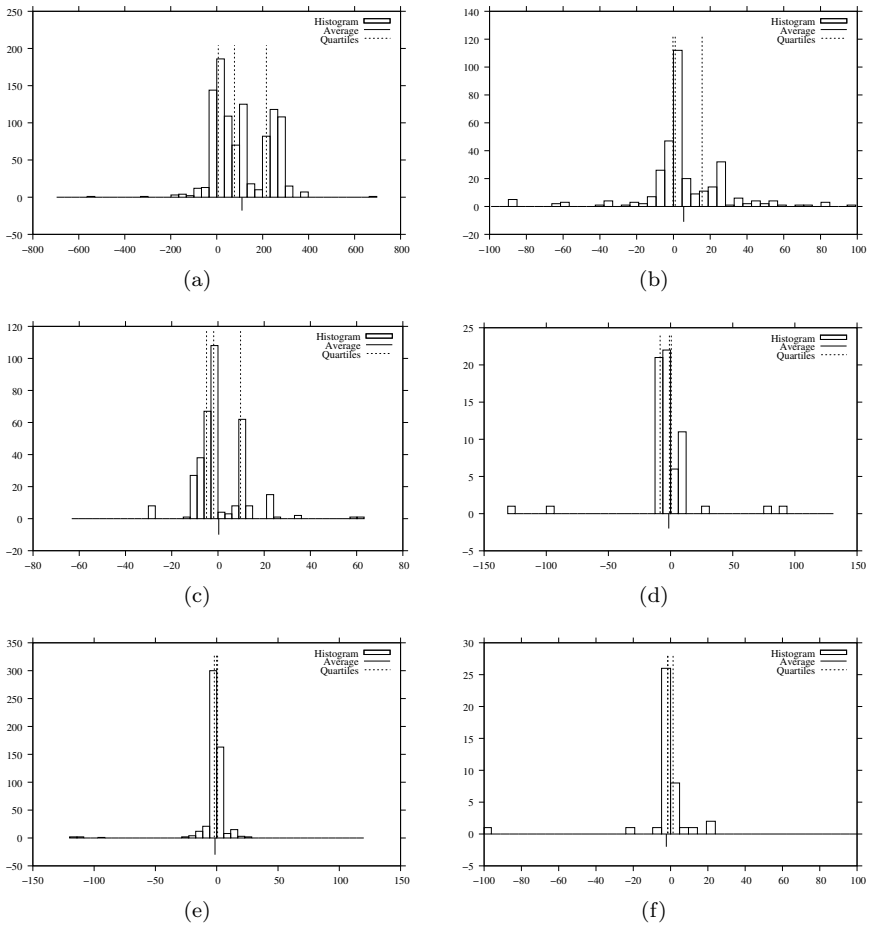

Figure 2: Histograms showing the number of jobs against the starting time difference (in hours) in each experiment. A positive value means that the task was executed earlier with DEMT than with FCFS.

impressively high, and we believe it is interesting to give an explanation. This part of the trace comes from the beginning of the usage of OAR on Icluster2, and the implementation of the FCFS scheduling algorithm was not exactly the same as the current one. Furthermore, this was a period a higher activity than the rest of the traces (the backlog was very large, and lots of jobs have a waiting time of about 10 days), and a large proportion of the jobs made used of properties and reservations, resulting in a much more constrained scheduling instance. In this context, where the natural bias towards FCFS is removed, and the instance is a very difficult one, our DEMT implementation performs much better than FCFS, mostly by delaying some larger tasks to make room for smaller and more constrained ones, and also by reordering the tasks so that compatible tasks are scheduled together.

The next step for experimentation will be to actually use this DEMT implementation in a live system. This will allow to see how the users react to a new scheduling policy, and also to conduct experiments biased in the other way. We also plan to convert classical traces from non-OAR clusters [7], and see the performance of both algorithms on these different instances, even though they come from systems with slightly different platform models.

6 Conclusion

This paper intended to bridge the gap from theoretical studies of scheduling to actual implementation environments. This work has allowed to outline in details the conceptual divergences between the classical models and the reality of actual scheduling environments. Furthermore, it has resulted in the implementation in such an environment of an algorithm based on theoretical foundations. The results are encouraging, and we hope that this will lead to more coordinated experiments between these two fields. For example, a theoretical study about scheduling parallel tasks in the presence of reservations is currently being conducted as a result of this practical work. Reciprocally, the next version of OAR, which is currently in a design phase, will include a support for moldable tasks.

References

[1] The grid'5000 project website. `http://www.grid5000.fr`.

[2] The icluster2 website. `http://www.inrialpes.fr/i-cluster2/`.

[3] The top500 organization website. `http://www.top500.org`.

[4] M. Baker, G. Fox, and H. Yau. Cluster computing review, 1995.

[5] Nicolas Capit, Georges Da Costa, Yiannis Georgiou, Guillaume Huard, Cyrille Marti n, Grégory Mounié, Pierre Neyron, and Olivier Richard. A batch scheduler with high level components. In *Cluster computing and Grid 2005 (CCGrid05)*, 2005.

[6] D. E. Culler, J. P. Singh, and A. Gupta. *Parallel Computer Architecture: A Hardware/Software Approach*. Morgan Kaufmann Publishers, inc., San Francisco, CA, 1999.

[7] Allen B. Downey and Dror G. Feitelson. The elusive goal of workload characterization. *Performance Evaluation Rev.*, 26(4):14–29, Mar 1999.

[8] M. Drozdowski. *Handbook of Scheduling*, chapter Scheduling Parallel Tasks - Algorithms and Complexity. CRC Press, 2004. chapter 27 of this book.

[9] P.-F. Dutot. *Algorithmes d'ordonnancement pour les nouveaux supports d'exécution*. PhD thesis, INP Grenoble, August 2004.

[10] P.-F. Dutot, L. Eyraud, G. Mounié, and D. Trystram. Bi-criteria algorithm for scheduling jobs on cluster platforms. In *Symposium on Parallel Algorithm and Architectures*, pages 125–132, Barcelona, 2004.

[11] P.-F. Dutot, G. Mounié, and D. Trystram. *Handbook of Scheduling*, chapter Scheduling Parallel Tasks - Approximation Algorithms. CRC Press, 2004. chapter 28 of this book.

[12] P.-F. Dutot and D. Trystram. A best-compromise bicriteria scheduling algorithm for malleable tasks. Research report (submitted to a journal).

[13] D. G. Feitelson. Scheduling parallel jobs on clusters. In Rajkumar Buyya, editor, *High Performance Cluster Computing*, volume 1, Architectures and Systems, pages 519–533. Prentice Hall PTR, Upper Saddle River, NJ, 1999. Chap. 21.

[14] Ian Foster and Carl Kesselman. *The grid: blueprint for a new computing infrastructure*. Morgan Kaufmann Publishers Inc., 1999.

[15] M. R. Garey and R. L. Graham. Bounds on multiprocessor scheduling with resource constraints. *SIAM Journal on Computing*, 4:187–200, 1975.

[16] C. Y. Lee. *Handbook of Scheduling*, chapter Machine Scheduling with Availability Constraints. CRC Press, 2004. chapter 22 of this book.

[17] J. Leung. *Handbook of Scheduling*, chapter Introduction and Notation. CRC Press, 2004. chapter 1 of this book.

[18] M. Mastrolilli. Scheduling to minimize max flow time: Off-line and on-line algorithms. *International Journal of Foundations of Computer Science*, 15:385–401, 2004.

[19] Grégory Mounié, Christophe Rapine, and Denis Trystram. Efficient approximation algorithms for scheduling malleable tasks. In *Eleventh ACM Symposium on Parallel Algorithms and Architectures (SPAA'99)*, pages 23–32. ACM, juin 1999.