Evaluation of Reallocation Heuristics for Moldable Tasks in Computational Grids

Yves Caniou[1,2,3] **Ghislain Charrier**[1,4] **Frédéric Desprez**[1,4]

[1]Université de Lyon, [2]UCBL, [3]CNRS (Jfli), [4]INRIA
Laboratoire de l'Informatique du Parallélisme (LIP), ÉNS Lyon
46 allée d'Italie, 69364 Lyon Cedex 07, FRANCE
Email: {yves.caniou, ghislain.charrier, frederic.desprez}@ens-lyon.fr

Abstract

Grid services often consist of remote sequential or rigid parallel application executions. However, moldable parallel applications, linear algebra solvers for example, are of great interest but requires dynamic tuning which has mostly to be done interactively if performances are needed. Thus, their grid execution depends on a *remote and transparent* submission to a possibly different batch scheduler on each site, and means an *automatic tuning* of the job according to the local load.

In this paper we study the benefits of having a middleware able to automatically submit and reallocate requests from one site to another when it is also able to configure the services by tuning their number of processors and their walltime. In this context, we evaluate the benefits of such mechanisms on two multi-cluster Grid setups, where the platform is either composed of several heterogeneous dedicated clusters, or non dedicated ones. Different scenarios are explored using simulations of real cluster traces from different origins.

Results show that a simple method is good and often the best. Indeed, it is faster and thus can take more jobs into account while having a small execution time. Moreover, users can expect more jobs finishing sooner and a gain on the average job response time between 10% and 40% in most cases if this reallocation mechanism combined to auto-tuning capabilities is implemented in a Grid framework. The implementation and the maintenance of this heuristic coupled to the migration mechanism in a Grid middleware is also simpler because less transfers are involved.

Keywords: batch schedulers; computational grids; meta-schedulers; moldable tasks; reallocation

1 Introduction

In order to meet the evergrowing needs in computing capabilities of scientists of all horizons, new computing paradigms have been explored including the Grid. The Grid is the aggregation of heterogeneous computing resources connected through high speed wide area networks.

This work has been supported in part by the ANR project SPADES (08-ANR-SEGI-025).

Copyright ©2011, Australian Computer Society, Inc. This paper appeared at the 9th Australian Symposium on Parallel and Distributed Computing (AusPDC 2011), Perth, Australia. Conferences in Research and Practice in Information Technology (CRPIT), Vol. 118, J. Chen and R. Ranjan, Ed. Reproduction for academic, not-for profit purposes permitted provided this text is included.

Computing resources are often parallel architectures managed by a local resource manager, called batch scheduler. In such a case, the local submission of a job requires at least a number of processors and a walltime. The walltime is the expected execution time for this job, given by the user or computed using data mining techniques. In most local resource management systems, when the walltime is reached, the job is killed, so users tend to over-evaluate the walltime to be sure that their job finishes its execution. Furthermore, giving an estimation of the execution time of a job is not an easy task and is influenced by the number of processors, which is generally chosen depending on external parameters such as the cluster load. Errors made at the local resource level may have a great impact on the global scheduling as shown by Beltrán & Guzmán (2009). Errors can come from mistakes on the walltime as well as a burst of submission as shown by Sonmez et al. (2009). Thus, having a mechanism to accommodate bad scheduling decisions is important.

The context of this work, described in detail by Caniou et al. (2010*b*), takes place in a heterogeneous multi-cluster Grid connected through a high bandwidth network: We propose a reallocation mechanism that takes into account scheduling errors by moving waiting jobs between clusters. The mechanism we propose can be used to connect different clusters together while each cluster keeps its local scheduling or resource allocation policies. Each job submitted onto the platform is executed automatically without any intervention from the user.

Two reallocation algorithms are studied with two heuristics each. We evaluate each couple (algorithm, heuristic) by comparing them on different metrics to an execution where reallocation is not performed. We extend the simulations realized by Caniou et al. (2010*a*) by focusing on moldable tasks instead of parallel rigid tasks. The middleware is able to determine the number of processors and the walltime automatically for each task. Furthermore, we study the algorithms on dedicated platforms as well as non dedicated platforms. *We aim at showing the expectations in terms of performance with regard to the increased complexity of the jobs management done by the middleware.* We analyze the results on different metrics, and we show that obtained gains are very good in the majority of the simulations we perform. Gains are larger on dedicated platforms than on non dedicated platforms. We show that in most cases reallocating jobs will let jobs to finish sooner and diminish their average response time between 10% and 40%. Furthermore, results definitely confirm the counter intuitive fact that even for moldable jobs, whose number of processors varies if migrated, the simplest heuristic, both algorithmically and in implementation complexity, is the best to use. Results presented in this work are only for heterogeneous platforms. A

more complete analysis, with results for both homogeneous and heterogeneous platforms, each with different batch scheduler policies, is available in research report (Caniou et al. 2010*b*).

The remainder of the paper is as follows. In Section 2, we present related work. In Section 3 we describe mechanisms and the scheduling algorithms used in this work. Then we explain the experimental framework in Section 4, giving information about the simulator we developed, on the platforms simulated with real-world traces, scenarios of experiments that were conducted as well as the metrics on which results are compared in Section 5. Finally we conclude in Section 6.

2 Background

Parallel applications are characterized by Feitelson et al. (1997) as rigid, moldable or malleable. A rigid application has a fixed number of processors. A moldable application can be executed with different number of processors, but once the execution started, this number can not change. Finally, the most permissive applications are malleable. The number of processors used can be modified "on the fly" during execution.

Cirne & Berman (2002) use moldable jobs to improve the performance in supercomputers. The user provides the scheduler *SA* with a set of possible requests that can be used to schedule a job. Such a request is represented by a number of processors and a walltime. SA chooses the request providing the earliest finish time. The evaluation of SA is done using real traces from the Parallel Workload Archive and their results show an average improvement on the response time of 44%, thus justifying the use of moldable jobs instead of rigid ones. In our work, we use the same kind of technique to choose the number of processors and the walltime of jobs. However, the user does not provide any information. The middleware is able to automatize everything thus facilitating the user's actions and can choose to migrate jobs from on site to another one.

Guim & Corbalán (2008) present a study of different meta-scheduling policies where each task uses its own meta-scheduler to be mapped on a parallel resource. Once submitted, a task is managed by the local scheduler and is never reallocated. In order to take advantage of the multi-site environment considered in our work, we use a central meta-scheduler to select a cluster for each incoming task because we place ourselves in the GridRPC context where clients do not know the computing resources. Also, once a task is submitted to the local scheduler, our approach let us cancel it and resubmit it elsewhere.

Yue (2004) presents the Grid-Backfilling. Each cluster sends a snapshot of its state to a central scheduler at fixed intervals. Then the central scheduler tries to back-fill jobs in the queue of other clusters. The computation done by the central scheduler is enormous since it works with the Gantt chart of all sites. All clusters are homogeneous in power and size. In our work, the central scheduler is called upon arrival of each job in order to balance the load among clusters. During the reallocation phase, it gathers the list of all the waiting tasks and asks the local schedulers when a job would complete, but it does not perform complex computations. Furthermore, in our work, clusters are heterogeneous in size and power and we consider moldable jobs.

Huang et al. (2009) present a study of the benefits of using moldable jobs in an heterogeneous computational grid. In this paper, the authors show that using a Grid meta-scheduler to choose on which site to execute a job coupled with local resource management schedulers able to cope with the moldability of jobs improves the average response time. In our work, instead of letting the local schedulers decide of the number of processors for a job, we keep existing infrastructure and software and we add a middleware layer that takes the moldability into account. Thus, our architecture can be deployed in existing Grids without modifications of the existing. Furthermore, this middleware layer renders reallocation between sites possible.

3 Task Reallocation

In this section, we describe the proposed tasks reallocation mechanism. First, we present the architecture of the Grid middleware (Section 3.1). Then we present the different algorithms used for the tasks reallocation (Section 3.2).

3.1 Architecture of the Middleware

Caniou et al. (2010*b*) describe the architecture that we use in this work. It is very close to the GridRPC (Seymour et al. 2004) standard from the Open Grid Forum[1]. Thus it can be implemented in GridRPC compliant middleware such as DIET (Caron & Desprez 2006) or Ninf (Sato et al. 1997). Because such a middleware is deployed on existing resources and has limited possibilities of action on the local resource managers, we developed a mechanism that only uses simple queries such as submission, cancellation, and estimation of the completion time.

The architecture relies on three main components: the **client** has computing requests to execute, and contacts an **agent** in order to obtain the reference of a **server** able to process the request. In our proposed architecture, one server is deployed on the front-end of each parallel resource, in which case it is in charge of interacting with the batch scheduler to perform the submission, cancellation or estimation of the completion date of a job. The server is also in charge of deciding how many processors should be used to execute the request, taking into account the load of the parallel resource. Benefiting from servers estimations, the *agent maps every incoming requests using a MCT strategy* (Minimum Completion Time (Maheswaran et al. 1999)), and *decides of the reallocation with a second scheduling heuristic.*

The process of submission of a job is depicted in Figure 1. 1) When a client wants to execute a request, it contacts the agent. 2) The agent then contacts each server where the service is available. 3) Each server able to execute the request computes an estimation of the completion time and 4) sends it back to the agent. 5) The agent sends the identity of the best server to the client which then 6) submits its request to the chosen server. 7) Finally, the server submits the task to the batch scheduler of the cluster. 8) When the agent orders a server to reallocate a task, the latter submits it to the other server provided by the agent.

3.2 Algorithms

This section presents the algorithm used to decide of the number of processors and walltime for each task (Section 3.2.1), the two versions of the reallocation mechanism (Section 3.2.2), and the scheduling heuristics used for reallocation (Section 3.2.3).

[1] http://www.ogf.org

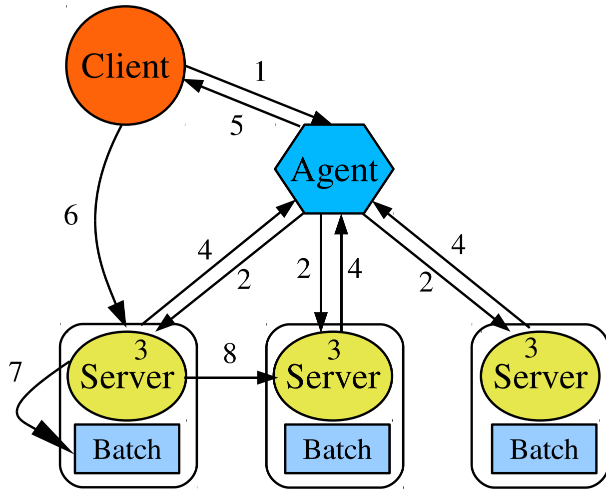

Figure 1: Architecture of the middleware layer for reallocation.

3.2.1 Tuning Parallel Jobs at Submission Time

The choice of the number of processors and walltime is done by the server each time a request arrives, either for the submission of a job or for an estimation of completion time. To determine the number of processors to allocate to the job, the server performs several estimations with different number of processors and returns the best size, *i.e.,* the one giving the earliest completion time. To estimate the completion time, the server can directly query the batch scheduler (but this capability is generally not present) or have it's own mechanism to compute the estimated completion time by simulating the batch algorithm for example.

The simplest idea to obtain the best size for the job is to perform an *exhaustive search*: For all possible number of processors (from one to the number of processors of the cluster), the estimation method provides a completion time as regard to the current load of the cluster. This method is simple and will choose the best size for jobs, however, it is time consuming. Indeed, each estimation is not instantaneous. Thus, for a large cluster, the estimation must be done a lot of times and the finding of the number of processors can require a long time.

Sudarsan & Ribbens (2010) benchmark different sizes of the LU application from the NAS parallel benchmarks[2]. Their study show a strictly increasing speedup up to 32 processors (adding processors always decreases execution time). But after this point, the execution time increases. It is due to the computation to communication ratio of the job becoming too small. This kind of job is not uncommon, thus we consider moldable jobs with *strictly increasing speedups until a known number of processors.*

Thus, in order to improve the speed in choosing the number of processors of a task, we can restrict the estimation from one processor to the limit of processors of the job. For jobs that don't scale very well, this will greatly reduce the number of calls to the estimation method thus reducing the time needed to find the most suitable number of processors.

Because of the hypothesis that speedup is strictly increasing until a maximum number of processors, we propose to perform a *binary search* on the number of processors to find how many of them to allocate to the job. Instead of estimating the completion time for each possible number of processors, we start by estimating the time for 1 processor and for the maximum number of processors. Then, we perform a classical binary search on the number of processors. This reduces the number of estimations from n to $log_2 n$.

In particular cases the binary search will not provide the optimal result because of the back-filling. Let us consider an example in order to illustrate this behavior. Consider a cluster of 5 processors and a job needing 7 minutes to be executed on a single processor. With a perfect parallelism, this jobs needs 3.5 minutes to run on 2 processors, 2.33 on 3, 1.75 on 4 and 1.4 on 5. Upon submission, the cluster has the load represented by hatched rectangles in Figure 2. First, the binary search evaluates the completion time for the job on 1 and 5 processors (top of the figure) and obtains completion times of 7 and 7.4 minutes respectively. Then, the number of processors is set to 3 (middle of 1 and 5). The evaluation returns a completion time of 7.33 (bottom left of the figure). The most promising completion time was obtained with 1 processor, thus the binary search looks between 1 and 3. Finally, the best completion for the tested values time is obtained for 2 processors: 6.5 minutes (bottom right). However, the best possible completion time the job could have is 1.75 minutes with 4 processors. Indeed, with 4 processors, the jobs can start as soon as submitted, but this value was disregarded by the binary search. During our tests to verify the behavior of the binary search on thousands jobs, the results were the same as the exhaustive search which means that the "bad" cases are rare.

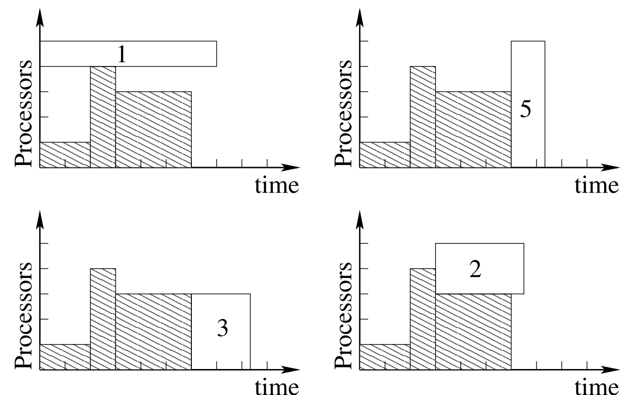

Figure 2: Estimations made by the binary search.

If the maximum number of processors of a job is large, using the binary search reduces enormously the number of estimations to do, potentially by orders of magnitude. For example, if a job can be executed on 650 processors the exhaustive search performs 650 estimations of completion time and the binary search performs only 10. The binary search in this case is thus 65 times faster.

3.2.2 Reallocation Algorithms

The first algorithm, **regular**, works as follows: It gathers the list of all jobs in the waiting queues of all clusters; it selects a job with a scheduling heuristic; if it is possible to submit the job somewhere else with a better estimated completion time (ECT) of at least a minute, it submits it on the other cluster and cancels the job at its current location; finally, it starts again with the remaining jobs. The one minute threshold is here to consider some small data transfer that can take place, and to diminish the number of reallocations bringing almost no improvement.

To have a better idea of what is done, consider an example of two batch systems with different loads (see

[2] http://www.nas.nasa.gov/Resources/Software/npb.html

Figure 3). At time t, task f finishes before its walltime, thus releasing resources. Task j is then scheduled earlier by the local batch scheduler. When a reallocation event is triggered by the meta-scheduler at t_1, it reallocates tasks h and i to the second batch system because their expected completion time is better there. To reallocate the tasks, each one is sequentially submitted to the second batch and canceled on the first one. In this example, the two clusters are identical so the tasks have the same execution time on both clusters, and the tuning of the parallel jobs (choice of number of processors to allocate to task h and i) is the same due to the same load condition. In an heterogeneous context, the length and even the number of processors allocated to the tasks would change between the clusters. Note that a task starting earlier on a cluster does not imply that it will also finish earlier.

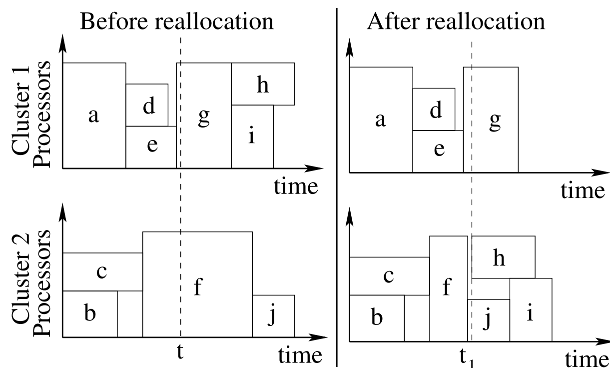

Figure 3: Example of reallocation between two clusters.

The second algorithm, **all-cancellation**, starts by canceling all waiting jobs of all clusters. The agent keeps a reference for all jobs. Then, it selects a job with a scheduling heuristic. Finally, it submits the job to the cluster giving the minimum estimated completion time and loops on each of the remaining jobs.

Note that it does not mean that all parallel jobs will be tuned in the maximum of their performance since platforms are not necessarily dedicated to the Grid middleware, each cluster has its own load. It may be better to use less resources, thus have a longer execution time, but start earlier.

The reallocation event in both versions of the algorithm is triggered periodically every hour, based on previous works conducted by Caniou et al. (2009) where a smaller period did not change the results but required more network transfers and potentially more reallocations.

Because both reallocation algorithms use an estimation of the completion time, it is mandatory that clusters use a batch scheduling algorithm able to give some guaranty on the completion time to guaranty the results. Feitelson et al. (2004) present the two main algorithms offering these guaranties are First-Come-First-Served (FCFS) and Conservative Back-Filling (CBF). Both algorithms make reservations for each job and jobs can never be delayed once the reservation done. However, jobs can be scheduled earlier if new resources become available. Batch schedulers using one of these algorithms are common. Other algorithms such as Easy Back-Filling (EBF) introduced by Lifka (1995) or the well-known Shortest Job First (SJF) presented by Feitelson et al. (1997) do not guaranty a completion time and thus should not be used without adding specialized prediction mechanisms to the servers.

3.2.3 Scheduling Heuristics for Reallocation

We focus on two heuristics to use to select a job at each iteration. With the first one, jobs are processed in their submission order. In the remainder of the paper, we refer to this policy as MCT because jobs are submitted in their original submission order and the jobs are submitted to the cluster with the Minimum Completion Time (MCT).

The second policy executes the MinMin heuristic on a subset of the jobs. MinMin asks the estimated completion time of all jobs and selects the job with the minimum of the returned values. In this paper, MinMin is executed on the 20 oldest jobs. We use this limit to avoid a too long reallocation time. Indeed, MinMin has to update the estimations of completion times of all the remaining jobs at each iteration to select the job with the minimum of the ECTs. Because the all-cancellation algorithm needs to resubmit all jobs, it executes MinMin on the 20 oldest jobs and then the remaining jobs are processed in their original submission order, leading to a MCT policy.

We have two scheduling heuristics, MCT and MinMin, as well as two reallocation algorithms, namely regular and all-cancellation. Thus, we have four couples of algorithm that we refer in the remainder of this paper as *MCT-reg*, *MCT-can*, *MinMin-reg*, and *MinMin-can*.

4 Experimental Framework

In this section we depict the experimental framework by presenting the simulator we implemented to run our experiments (Section 4.1), the description of the jobs (Section 4.2), the simulated platforms (Section 4.3), and the metrics used to compare the heuristics (Section 4.4). Finally, the experiments are described (Section 4.5).

4.1 Simulator

In order to simulate task reallocation in a distributed environment composed of several clusters, we use SimGrid (Casanova et al. 2008), a discrete events simulation toolkit designed to simulate distributed environments, and Simbatch (Caniou & Gay 2009), a batch systems simulator built on top of SimGrid. Simbatch, which has been tested against real life experiments, can simulate the main algorithms used in batch schedulers described by Feitelson et al. (2004). In this study, we use the *Conservative Back-Filling* (CBF) algorithm for the batch schedulers. Mu'alem & Feitelson (2001) introduces the CBF algorithm. It tries to find a slot in the queue (Back-filling) where the job can fit without delaying already scheduled jobs (Conservative). If it does not, the job is added at the end of the queue. CBF is available in batch systems such as Maui (Jackson et al. 2001), Loadleveler (Kannan et al. 2001), and OAR (Capit et al. 2005) among others.

The simulator is divided using the same components as the ones in the GridRPC standard introduced in Section 3.1:

The *client* requests the system for a service execution. It contacts the meta-scheduler that will answer with the reference of a server providing the desired service.

The *meta-scheduler* matches incoming requests to a server according to a scheduling heuristic (we use MCT in this paper) and periodically reallocates jobs in waiting queues on the platform using one of the reallocation scheduling heuristic described in Section 3.2.3.

The *server* is running on the front-end of a cluster and interacts with the batch system. It receives requests from the client and can submit jobs to the batch scheduler to execute the requests. It can also cancel a waiting job, return an estimation of the completion time of a request and return the list of jobs in the waiting state. For submission and estimation, the server uses an estimation function that automatically chooses the number of processors and the walltime of the request using the technique described in Section 3.2.1.

4.2 Jobs

We built seven scenarios of jobs submission, where for six of them, jobs come from traces of different clusters on GRID'5000 for the first six months of 2008. Table 1 gives the number of jobs per month on each cluster. The seventh scenario is a six month long simulation using two traces from the parallel workload archive (CTC and SDSC) and the trace of Bordeaux on GRID'5000. The trace from Bordeaux contains 74647 jobs, CTC has 42873 jobs and SDSC contains 15615 jobs. Thus, there is a total of *133135 jobs*. In the remainder of the paper, we refer at the different scenarios by the name of the month of the trace for the jobs from GRID'5000 and we refer to the jobs coming from CTC, SDSC, and GRID'5000 as "PWA-G5K".

Month/Cluster	Bordeaux	Lyon	Toulouse	Total
January	13084	583	488	14155
February	5822	2695	1123	9640
March	11673	8315	949	20937
April	33250	1330	1461	36041
May	6765	2179	1573	10517
June	4094	3540	1548	9182

Table 1: Number of jobs per month and in total for each site trace.

In our simulations, we do not consider advance reservations (present in GRID'5000 traces). They are considered as simple submissions so the batch scheduler can start them when it decides to. To evaluate the heuristics, we compare simulations together so this modification does not impact the results. However, we can not compare ourselves with what happened in reality. Furthermore, note that we add a meta-scheduler to map the jobs onto clusters at submission time, as if a grid middleware is used. On the real platform, users submit the cluster of their choice (usually they submit to the site closest to them) so the simulations already diverge from reality.

The traces taken from the Parallel Workload Archive were taken in their standard original format, *i.e.,* they also contain "bad" jobs described by Feitelson & Tsafrir (2006). We want to reproduce the execution of jobs on clusters, so we need to keep all the "bad" jobs removed in the clean version of the logs because these jobs were submitted in reality.

4.2.1 Moldable Jobs

The jobs contained in the trace files are parallel rigid jobs. So, in order to simulate the moldable jobs, we defined 4 types of jobs using Amdahl's law ($speedup = \frac{1}{(1-P)+\frac{P}{N}}$ with P the fraction of parallel code and N the number of processors). The law states that the expected speedup of an application is strictly increasing but the increase rate diminishes. The execution time of an application tends to the execution time of the sequential portion of the application when adding more processors.

To obtain the 4 types of moldable jobs, we vary the parallel portion of the jobs that is sequential as well as the limit of processors until the execution time decrease. The different values for the parallel portion of code are 0.8, 0.9, 0.99 and 0.999. Figure 4 plots the speedups for the different values of parallel code for different number of processors. Note that the y-axis is log-scaled. The figure shows that there is some point where the speedup increase becomes negligible. For the limits, we chose to use 32, 96, 256, and 650 processors. These values were chosen in accordance to the gain on the execution time of adding one processor. When the gain becomes really small, chances are that the internal communications of the job will take most of the time and slow down the task. Furthermore, the 650 limit is given by the size of the largest cluster of our simulations. So, the 4 types of jobs we consider are 4 couples (parallel portion, limit of processors): $t1$:(0.8, 32), $t2$:(0.9, 96), $t3$:(0.99, 256) and $t4$:(0.999, 650).

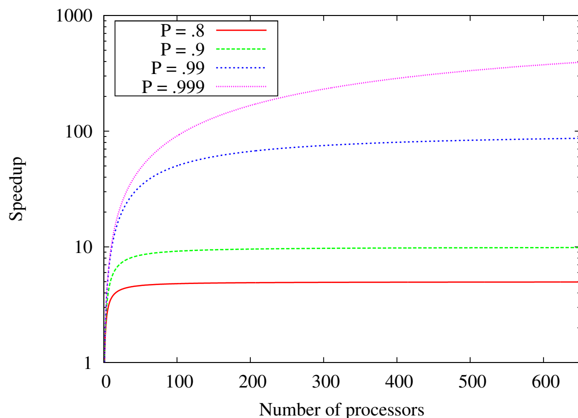

Figure 4: Speedups for the Amdahl's law for different parallelism portions.

In the traces, there are more tasks using a small number of processors than tasks using a lot of processors. Thus, each job from the trace files was given a moldable type. In each simulation we present, there are 50% of jobs of type $t1$, 30% of type $t2$, 15% of type $t3$ and 5% of type $t4$. The type of a job is chosen randomly. In order to keep a more realistic set of jobs, we decided to keep the sequential jobs of the traces sequential.

4.2.2 Simulating Realistic Parallel Jobs

During the simulations, the server uses information from both the traces and the type of the job to choose a suitable number of processors and a walltime for the job. In order to do so, the server uses the binary search described in Section 3.2.1 to choose a number of processors and follows the following process to choose the walltime: First, it computes the speedup of the job in the trace file using Amdahl's law, the type of the job and the number of processors: $spd = amdahl(p, n_t)$ with p the parallel portion of the code and n_t the number of processors used in the trace file. Second, the server computes the walltime of the job on one processor: $w_1 = w_{n_t} * spd$. Third, the server computes the speedup of the job for the current number of processors chosen by the binary search: $spd_b = amdahl(p, n_b)$. Then, the server computes the walltime for the job: $w_b = \frac{w_1}{spd_b}$. Finally, the runtime and walltime are modified *in accordance with the speed of the cluster* given in Section 4.3.1.

To obtain the actual execution time for the moldable jobs, we keep the same difference ratio as the

one in the trace file. Thus if the runtime of a job was twice smaller than walltime in the trace file, it will also be twice smaller than the walltime in the simulations, independently of the number of processors chosen for the job.

4.3 Platform Characteristics

4.3.1 Computing Resources

We consider two sets of resources, composed of three sites, each with a different number of cores, and managed with a CBF policy.

The first set corresponds to the simulation of three clusters of Grid'5000 (Bolze et al. 2006). The three clusters are Bordeaux, Lyon, and Toulouse. Bordeaux is composed of 640 cores and is the slowest cluster. Lyon has 270 cores and is 20% faster than Bordeaux. Finally, Toulouse has 434 cores and is 40% faster than Bordeaux.

The second set corresponds to experiments mixing the trace of Bordeaux from Grid'5000 and two traces from the Parallel Workload Archive[3]. The three clusters are Bordeaux, CTC, and SDSC. Bordeaux has 640 cores and is the slowest cluster. CTC has 430 cores and is 20% faster than Bordeaux. Finally, SDSC has 128 cores and is 40% faster than Bordeaux.

4.3.2 Dedicated Vs. Non Dedicated

On real life sites, tasks can be either submitted by a Grid middleware or by local users. Thus, we investigate the differences in behavior of our mechanism depending on heuristics: on **dedicated platforms**, where all tasks have been submitted through our middleware; on **non dedicated platforms** where two third of the jobs issued from the traces are directly submitted through batch schedulers by simulated local users. Both setups will be investigated in Sections 5.1 and 5.2 for the dedicated case and for the non dedicated platform respectively.

4.4 Evaluation Metrics

We choose the following metrics to compare the performance of reallocation depending on platforms, mechanisms and scheduling heuristics:

The percentage of jobs impacted by reallocation is the percentage of jobs whose completion time is changed compared to an execution without reallocation. In this study, we are only interested by these jobs.

We also study **the number of reallocations relative to the total number of jobs**. We give the percentage of reallocations in comparison of the number of jobs. A job can be counted several times if it migrated several times so it is theoretically possible to have more than 100% reallocations. A small value is better because it means less transfers.

On a user point of view, **the percentage of jobs finishing earlier with reallocation than without** is very important. This percentage is taken only from the jobs whose completion time changed with reallocation. A value higher that 50% means that there are more jobs early than late.

Feitelson & Rudolph (1998) presents the notion of response time. It corresponds to the duration between submission and completion. Complementary to the previous one, **the average job response time** of the jobs impacted by reallocation relatively to the scenario without reallocation defines the average ratio that the duration of a job can issue. A ratio of 0.8 means that on average, jobs spent 20% less time in the system, thus giving the results faster to the users.

[3]http://www.cs.huji.ac.il/labs/parallel/workload/

Figure 5 illustrates why jobs can be delayed and others finishing earlier onto a platform composed of two clusters. At time 0 a reallocation event is triggered. A task is reallocated from cluster 2 to cluster 1 with a greater number of processors allocated to it according to our algorithm. Thus, some tasks of cluster 2 are advanced in the schedule. On cluster 1, as expected, the task is back-filled. However, assume the task finishing at time 6 finishes at time 2 because the walltime was wrongly defined (see the task with the dashed line). Thus, because of the newly inserted task, the large task on cluster 1 is delayed. Note that, even with FCFS, reallocation can also cause delay. If a job is sent to a cluster, all the jobs submitted after may be delayed. Inversely, the job that was reallocated to another cluster now leaves some free space and it may be used by other jobs to diminish their completion time.

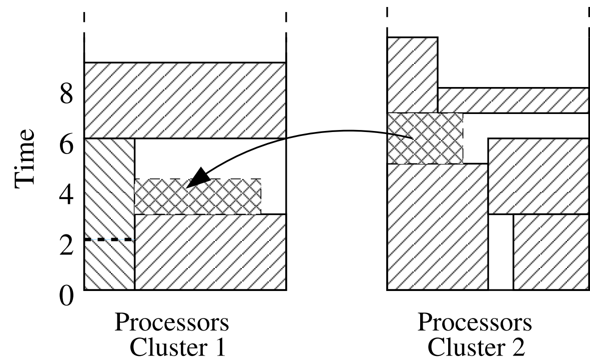

Figure 5: Side effects of a reallocation.

4.5 Experiment

An experiment is a tuple (reallocation algorithm, heuristic, platform-trace, dedicated, seed) where the seed is used to draw the type of a job in the trace, and concerning non dedicated platform, to draw if a job is submitted to the middleware or directly to the local scheduler. We used 10 different random seeds, hence, in addition to the reference experiment using MCT without reallocation, we conducted 14+2*2*7*2*10, *i.e.,* **574 experiments** in total.

5 Results

First, we present the results on dedicated platforms in Section 5.1. Then, Section 5.2 contains the results for non dedicated platforms. Finally, some concluding remarks on the results are given in Section 5.3.

Figures in this section show the minimum, the maximum, the median, the lower, and higher quartiles and the average of the 10 runs of each experiment. Concerning the figures in non dedicated platforms, results only take into account the jobs submitted to the Grid middleware. External jobs are not represented in the plots.

5.1 Dedicated Platforms

In this section, clusters are heterogeneous in number of processors and in speed (cf. Section 4.3). All requests are done to our Grid middleware, thus there are no local jobs submitted.

The percentage of jobs impacted is shown in Figure 6. In six experiments for the two traces March and June, extreme cases were almost 100% of the jobs that

were impacted by reallocation appear. This happens when the platform has a few phases with no job. If there are always jobs waiting, the reallocation is able to move jobs more often thus impacting a bigger portion of the jobs. Apart from these cases, the number of jobs impacted varies between the traces from 25 to 95%. All-cancellation algorithms usually impacts more jobs. MinMin-can impacts more jobs on average than the other heuristics. MCT-reg and MinMin-reg have close results.

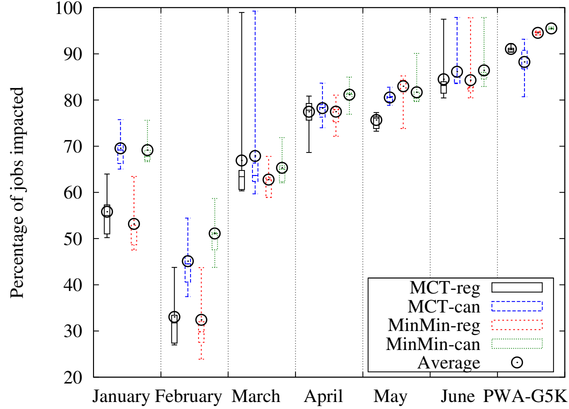

Figure 6: Jobs impacted on dedicated platforms.

The number of reallocations relative to the total number of jobs is plotted in Figure 7. All-cancellation algorithms always produce more reallocations. The regular algorithms give results inferior than 15% so the number of reallocations is quite small compared to the total number of jobs. However, with the all-cancellation algorithms, it is possible to go to a value as high as 50%. Because all-cancellation empties the waiting queues, more jobs have the opportunity to be reallocated. With the regular algorithms, jobs close to execution have a very small chance of being reallocated. The regular version of the reallocation algorithm is better on this metric.

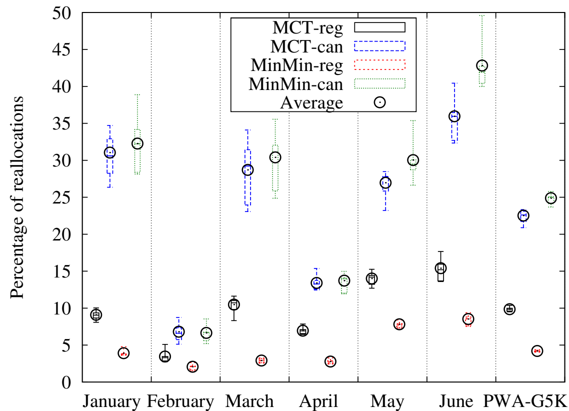

Figure 7: Reallocations on dedicated platforms.

Figure 8 plots the percentage of jobs early. In this case, 3 experiments produce more jobs late than early. In April without all-cancellation there are always more jobs late (less than 4%) when reallocation is performed. However in most cases, it is better to reallocate. MinMin-reg gives the worst results. It is followed by MCT-reg, then MinMin-can and finally MCT-can is the best with up to 64% of tasks early!

Concerning the average relative response time, the plot in Figure 9 shows a clear improvement in most

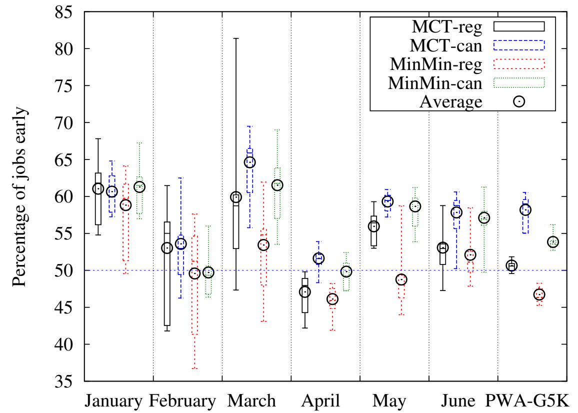

Figure 8: Percentage of jobs early on dedicated platforms.

cases. Excluding MinMin-reg, most gains are comprised between 10% and 40%. On average, MCT-can is the best heuristic. The reallocation without all-cancellation can worsen the average response time. It happened in 6 experiments (3 with MCT-reg and 3 with MinMin-reg). The loss is small for MCT-reg (less than 5%) thus it is not a problem. The all-cancellation versions are always better than their corresponding regular algorithm except in February for MCT-reg. Some experiments present a gain on the average response time while there were more jobs late than early (MCT-reg in April for example): The gains were high enough to compensate for the late jobs.

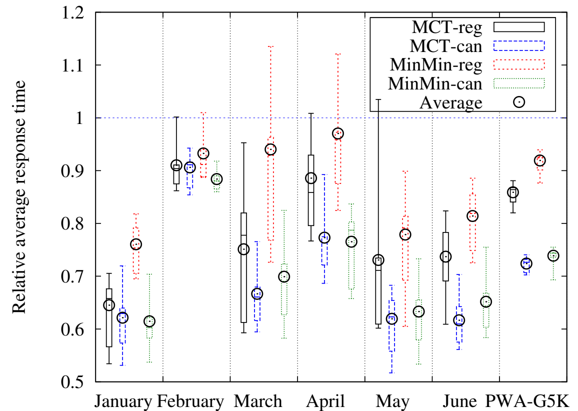

Figure 9: Relative average response time on dedicated platforms.

5.2 Non Dedicated Platforms

In this section, we present the results on non dedicated platforms where 33% of the jobs executed on the Grid platform are moldable and submitted to the Grid middleware.

The percentage of jobs impacted by reallocation is plotted in Figure 10. The two all-cancellation heuristics impact more jobs than the regular ones, but the difference is really small. There is one experiment in March where MinMin-reg impacts almost all jobs: a scheduling decision taken at the beginning of the experiment impacts all the following job completion dates. For a given trace, the number of impacted jobs usually does not vary a lot.

Figure 11 plots the number of reallocations relative to the total number of moldable jobs. The number of reallocations is very small. In most cases, there are

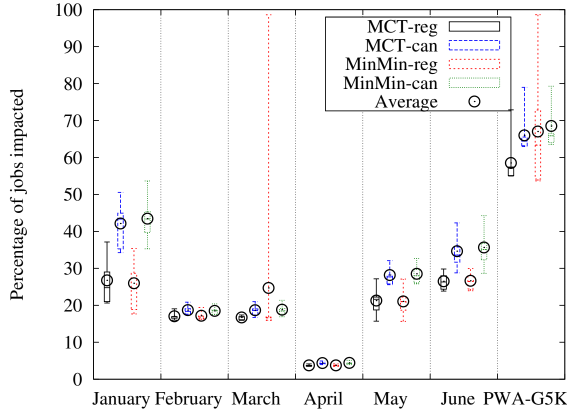

Figure 10: Jobs impacted on non dedicated platforms.

only a few dozens reallocations. The all-cancellation algorithms always reallocates more than the regular versions, *but not by far*. In a lot of cases, the number of reallocations corresponds to less than 1% of the number of jobs. Thus, on a non dedicated platform, the reallocation mechanism does not produce many transfers.

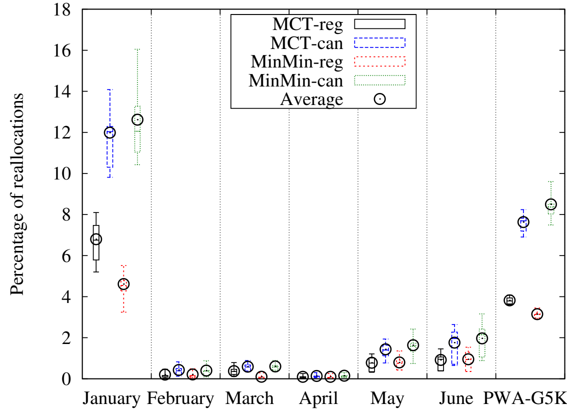

Figure 11: Reallocations on non dedicated platforms.

Most experiments except the worst case for PWA-G5K and March with MinMin-reg result *in more than half of the jobs early* as we can see in Figure 12. The 90% jobs late in March with MinMin-reg are from the same experiment where almost all jobs were impacted in Figure 10. Most experiments exhibit a percentage of jobs early close to 70%. All-cancellation again produces less jobs early than regular. MCT-reg and MinMin-reg are the two heuristics of choice, but MinMin-reg gives mitigate results for PWA-G5K so MCT-reg is a better choice.

Figure 13 shows that the different heuristics give results close to one another on the relative average response time. All-cancellation heuristics usually have a smaller difference between the minimum and the maximum gains. Depending on the experiment, results vary a lot. In some experiments, the average response time is divided by more than two, but in other it is augmented with a maximum of 40%. However on all experiments, the average gain is positive. Thus reallocation is expected to provide a gain.

5.3 Remarks on Results

MCT-reg and MinMin-reg usually give similar results on non dedicated platforms, often in favor of MCT-reg. On dedicated platforms however, MCT-reg is clearly better than MinMin-reg. MinMin-reg may give better results if it is able to take more jobs into account during reallocation. But, if it takes more jobs into account, its execution time grows exponentially. Furthermore, the two algorithms with all-cancellation also give very similar results with a small advantage for the MCT-reg on dedicated platforms.

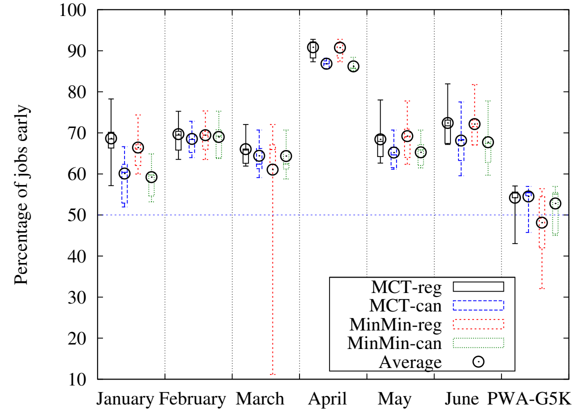

Figure 12: Percentage of jobs early on non dedicated platforms.

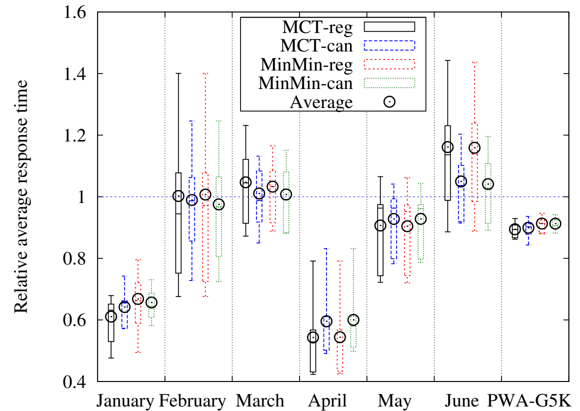

Figure 13: Relative average response time on non dedicated platforms.

The all-cancellation algorithms can cause starvation. In a non dedicated platform, it is obvious that starvation can happen. Indeed, when canceling jobs, jobs from the external load (for the Grid middleware) will be rescheduled in front of the moldable jobs managed by the middleware system. This may explain the worst cases peaks in Figure 13. Even in a dedicated environment with MinMin-can, it is possible for a job to be delayed indefinitely. If the job is long, it will always be resubmitted after others and may never start execution. However, such cases did not happen in our simulations because there are always phases of low load where the queues can be emptied.

The results presented in this paper show that the heuristic of choice is MCT with or without all-cancellation whether the platform is dedicated or not. Indeed, MinMin is too complex in time to react in a decent time regarding the submission rate of jobs onto the platform. In a previous study Caniou et al. (2010*a*), we used several other selection heuristics such as MaxMin, Sufferage, MaxGain, and MaxRelGain but these heuristic did not prove better than MCT or MinMin. Because these algorithms have the same complexity than MinMin, we argue that they may also give poor results, especially because of worst

cases.

In this paper, due to space constraints, we present only results on heterogeneous platforms, since they are the most common in real life. But results on homogeneous platforms are presented in detail in the research report (Caniou et al. 2010*b*), where clusters have different sizes, but their speed is the same. Gains obtained by the reallocation are usually better by a few percents on homogeneous platforms than the one presented in this paper. The same patterns as the ones we see in this paper emerge: on dedicated platform, MCT-can and MinMin-can give the best results. MCT-reg produces less gains, and the worst is MinMin-can. On non-dedicated platform, all heuristics give similar results.

6 Conclusion and Perspectives

In this paper, we presented a reallocation mechanism that can be implemented in a GridRPC middleware and used on any multi-cluster Grid without modifying the underlying infrastructure. Parallel jobs are tuned by the Grid middleware each time they are submitted to the local resource manager (which implies also each time a job is migrated). We achieve this goal by only querying batch schedulers with simple submission or cancellation requests. Users ask the middleware to execute some service and the middleware manages the job automatically.

We have investigated two reallocations algorithms, the key difference between them being that one, regular, cancels a task once it is sure that the expected completion time is better on another cluster, and the other, all-cancellation, cancels all waiting jobs before testing reallocation. We also considered two scheduling heuristics to make the decision of migrating a job to another site. We conducted 564 experiments and analyzed them on 4 different metrics.

On dedicated clusters, the cancellation of all the waiting jobs proves to be very efficient to improve the average job response time. On the other hand in an non dedicated environment, the algorithm that does not cancel waiting jobs behaves better. On both platforms, surprisingly, there is not a great number of migrating tasks, but all tasks take benefit of those migrations since the percentage of impacted tasks is high. In term of performances, users can expect more jobs finishing sooner, and an improvement of the jobs response time from a few percents to more than 50%! Only a few cases give bad results leading to an increase of the average job response time.

The next step of this work is the implementation of the reallocation mechanism in the DIET GridRPC middleware. DIET already provides most of the needed features. The missing features are the cancellation of a job in batch schedulers (numerous are supported) which is easy to implement and the reallocation mechanism itself. This last point should be quite straightforward because all communications are already handled by the middleware. We intend to implement both reallocation mechanisms with MCT. Indeed, we need the regular algorithm to work on non dedicated platforms. We plan also to implement the all-cancellation mechanism because DIET can be used in a dedicated environment. Furthermore, we could use this in the SPADES[4] project where we plan to maintain a set of reserved resources on a site which are managed by our own embedded batch scheduler.

[4] ANRProject08-ANR-SEGI-025

References

Beltrán, M. & Guzmán, A. (2009), 'The Impact of Workload Variability on Load Balancing Algorithms', *Scalable Computing: Practice and Experience* **10**(2), 131–146.

Bolze, R., Cappello, F., Caron, E., Daydé, M., Desprez, F., Jeannot, E., Jégou, Y., Lanteri, S., Leduc, J., Melab, N., Mornet, G., Namyst, R., Primet, P., Quetier, B., Richard, O., Talbi, E. & Touché, I. (2006), 'Grid'5000: A Large Scale and Highly Reconfigurable Experimental Grid Testbed', *International Journal of High Performance Computing Applications* **20**(4), 481–494.

Caniou, Y., Caron, E., Charrier, G. & Desprez, F. (2009), Meta-Scheduling and Task Reallocation in a Grid Environment, *in* 'The Third International Conference on Advanced Engineering Computing and Applications in Sciences (ADVCOMP'09)', Sliema, Malta, pp. 181–186.

Caniou, Y., Charrier, G. & Desprez, F. (2010*a*), Analysis of Tasks Reallocation in a Dedicated Grid Environment, *in* 'IEEE International Conference on Cluster Computing 2010 (Cluster 2010)', Heraklion, Crete, Greece. To appear.

Caniou, Y., Charrier, G. & Desprez, F. (2010*b*), Evaluation of Reallocation Heuristics for Moldable Tasks in Computational Dedicated and non Dedicated Grids, Technical Report RR-7365, Institut National de Recherche en Informatique et en Automatique (INRIA).

Caniou, Y. & Gay, J. (2009), Simbatch: An API for simulating and predicting the performance of parallel resources managed by batch systems, *in* 'Workshop on Secure, Trusted, Manageable and Controllable Grid Services (SGS), in conjunction with EuroPar'08', Vol. 5415 of *LNCS*, pp. 217–228.

Capit, N., Da Costa, G., Georgiou, Y., Huard, G., Martin, C., Mounié, G., Neyron, P. & Richard, O. (2005), 'A Batch Scheduler with High Level Components', *CoRR* p. 9.

Caron, E. & Desprez, F. (2006), 'DIET: A Scalable Toolbox to Build Network Enabled Servers on the Grid', *International Journal of High Performance Computing Applications* **20**(3), 335–352.

Casanova, H., Legrand, A. & Quinson, M. (2008), SimGrid: a Generic Framework for Large-Scale Distributed Experiments, *in* '10th IEEE International Conference on Computer Modeling and Simulation'.

Cirne, W. & Berman, F. (2002), 'Using Moldability to Improve the Performance of Supercomputer Jobs', *Journal of Parallel and Distributed Computing* **62**(10), 1571–1601.

Feitelson, D. & Rudolph, L. (1998), Metrics and Benchmarking for Parallel Job Scheduling, *in* 'Proceedings of the Workshop on Job Scheduling Strategies for Parallel Processing (IPPS/JSSPP '98)', Springer-Verlag, pp. 1–24.

Feitelson, D., Rudolph, L. & Schwiegelshohn, U. (2004), Parallel job scheduling - a status report, *in* 'Job Scheduling Strategies for Parallel Processing'.

Feitelson, D., Rudolph, L., Schwiegelshohn, U., Sevcik, K. & Wong, P. (1997), Theory and Practice in Parallel Job Scheduling, *in* 'IPPS '97: Proceedings of the Job Scheduling Strategies for Parallel Processing', Springer-Verlag, pp. 1–34.

Feitelson, D. & Tsafrir, D. (2006), Workload Sanitation for Performance Evaluation, *in* 'IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)', pp. 221–230.

Guim, F. & Corbalán, J. (2008), A Job Self-scheduling Policy for HPC Infrastructures, *in* 'Job Scheduling Strategies for Parallel Processing (JSSPP)'.

Huang, K., Shih, P. & Chung, Y. (2009), 'Adaptive Processor Allocation for Moldable Jobs in Computational Grid', *International Journal of Grid and High Performance Computing* **1**(1), 10–21.

Jackson, D., Snell, Q. & Clement, M. (2001), Core Algorithms of the Maui Scheduler, *in* 'JSSPP '01: Revised Papers from the 7th International Workshop on Job Scheduling Strategies for Parallel Processing', Springer-Verlag, pp. 87–102.

Kannan, S., Roberts, M., Mayes, P., Brelsford, D. & Skovira, J. (2001), *Workload Management with LoadLeveler*, IBM Press.

Lifka, D. (1995), The ANL/IBM SP scheduling system, *in* 'In Job Scheduling Strategies for Parallel Processing', Springer-Verlag, pp. 295–303.

Maheswaran, M., Ali, S., Siegel, H., Hensgen, D. & Freund, R. (1999), 'Dynamic Matching and Scheduling of a Class of Independent Tasks onto Heterogeneous Computing Systems', *Journal of Parallel and Distributed Computing* **59**, 107–131.

Mu'alem, A. & Feitelson, D. (2001), '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.

Sato, M., Nakada, H., Sekiguchi, S., Matsuoka, S., Nagashima, U. & Takagi, H. (1997), Ninf: A Network Based Information Library for Global World-Wide Computing Infrastructure, *in* 'HPCN Europe', pp. 491–502.

Seymour, K., Lee, C., Desprez, F., Nakada, H. & Tanaka, Y. (2004), The End-User and Middleware APIs for GridRPC, *in* 'Workshop on Grid Application Programming Interfaces, In conjunction with GGF12'.

Sonmez, O., Yigitbasi, N., Iosup, A. & Epema, D. (2009), Trace-Based Evaluation of Job Runtime and Queue Wait Time Predictions in Grids, *in* 'International Symposium on High Performance Distributed Computing (HPDC'09)'.

Sudarsan, R. & Ribbens, C. (2010), 'Design and performance of a scheduling framework for resizable parallel applications', *Parallel Computing* **36**, 48–64.

Yue, J. (2004), 'Global Backfilling Scheduling in Multiclusters', *Lecture notes in Computer Science* **3285**, 232–239.