Experiences on Adaptive Grid Scheduling of Parameter Sweep Applications *

Eduardo Huedo[†] Rubén S. Montero[‡] Ignacio M. Llorente[‡,†]

[†]Lab. Computación Avanzada
Centro de Astrobiología (CSIC-INTA)
28850 Torrejón de Ardoz (Spain)
{huedoce, martinli}@inta.es

[‡]Dpto. Arquitectura de Computadores y Automática
Facultad de Informática, Universidad Complutense
28040 Madrid (Spain)
{rubensm, llorente}@dacya.ucm.es

Abstract

*Grids offer a dramatic increase in the number of available compute and storage resources that can be delivered to applications. This new computational infrastructure provides a promising platform to execute loosely coupled, high-throughput parameter sweep applications. This kind of applications arises naturally in many scientific and engineering fields like Bioinformatics, Computational Fluid Dynamics (CFD), Particle Physics, etc. The efficient execution and scheduling of parameter sweep applications is challenging because of the dynamic and heterogeneous nature of Grids. In this paper we present a scheduling algorithm built on top of the Grid*Way *framework that combines: (i) adaptive scheduling to reflect the dynamic Grid characteristics; (ii) adaptive execution to migrate running jobs to* better *resources and provide fault tolerance; (iii) re-use of common files between tasks to reduce the file transfer overhead. The efficiency of the approach presented in this work is demonstrated in the execution of a CFD application on a highly heterogeneous research testbed.*

1. Introduction

Grid computing constitutes an appropriate platform to execute parameter sweep applications (PSAs), which arise naturally in many scientific and engineering fields like Bioinformatics, Computational Fluid Dynamics (CFD), Particle Physics, etc. This kind of applications comprises the execution of a high number of tasks, each of which performs a given calculation over a subset of parameter values. In the present work, we will consider a PSA as a set of *independent task*, that potentially share common files (e.g. the executable, or some input files). In spite of the relatively simple structure of the PSAs, its efficient execution on computational Grids involves challenging issues, mainly due to the nature of the Grid itself.

Probably, one of the most challenging problems that the Grid computing community has to deal with, to efficiently execute applications as the one described above, is the fact that Grids present unpredictable changing conditions, namely: high fault rate and dynamic resource availability, load and cost.

Adaptive Grid scheduling has been widely studied in the literature [2, 3, 1, 14] and is generally accepted as the cure to the dynamicity of the Grid. Previous works have clearly demonstrated the critical factor of the dynamic information gathered from the Grid to generate reliable schedules.

In this paper, we modify several components of the Grid*W*ay framework, which is briefly described in Section 2, to efficiently schedule PSAs. In order to achieve the adaptive functionality, we present in Section 3, a resource broker that reflects application preferences and the dynamic characteristics of Grid resources, in terms of load, availability and proximity.

In addition, the ability to migrate running jobs to more suitable resources based on events dynamically generated by both the Grid and the running applications (*adaptive execution*), can also improve the performance and fault tolerance obtained by applications on a Grid [9]. The support for adaptive execution of the Grid*W*ay framework is discussed in Section 4; and then, in Section 5, we describe the Grid*W*ay facilities to provide job execution with fault tolerance. Moreover, efficient execution of PSAs can only be achieved by re-using shared files between tasks [4]. This is specially important not only to reduce the file transfer overhead, but also to prevent saturation of the file server where these files are stored, which can occur with large-scale PSAs.

This three aspects, namely: adaptive scheduling, adaptive execution and re-use of shared files, are combined in Section 6 to devise a Grid scheduling policy for PSAs. Fi-

* This research was supported by Ministerio de Ciencia y Tecnología through the research grant TIC 2002-00334 and Instituto Nacional de Técnica Aeroespacial (INTA).

nally, in Section 7 we will show the benefits of the above components in the execution of CFD parameter sweep application. The schedule performed by the Grid*W*ay framework is also compared to the optimum schedule, using a standard performance metric: the application *makespan*. The paper ends with some conclusions.

2. The Grid*W*ay Framework

Grid*W*ay is a Globus-based experimental framework that allows an easier and more efficient execution of jobs on a dynamic Grid environment in a "submit and forget" fashion. The core of the Grid*W*ay framework is a personal *submission agent* that automatically performs the steps involved in job submission [13]: resource discovery and selection; and job preparation, submission, monitoring, migration and termination. Adaptation to changing conditions is achieved by supporting automatic job migration. Once the job is initially allocated, it is dynamically rescheduled when one of the following *rescheduling* events occurs:

1. ***Grid-generated rescheduling events:***
    - A new "better" resource is discovered (opportunistic migration [11]).
    - The remote resource or its network connection fails (failover migration).
    - The submitted job is canceled or suspended by the local resource administrator or management system.
2. ***Application-generated rescheduling events:***
    - Performance degradation or performance contract violation is detected in terms of application intrinsic metrics.
    - The resource requirements or preferences of the application change (self-migration).

The architecture of the *submission agent* is depicted in figure 1. The user interacts with the framework through a *request manager*, which handles client requests (*submit*, *kill*, *stop*, *resume*...) and forwards them to the *dispatch manager*. The *dispatch manager* periodically wakes up at each *scheduling* interval, and tries to submit pending and rescheduled jobs to Grid resources. Once a job is allocated to a resource, a *submission manager* and a *performance monitor* are started to watch over its correct and efficient execution (see [8] for a detailed description of these components).

The framework has been designed to be modular, thus allowing extensibility and improvement of its capabilities. The following modules can be set on a per job basis:

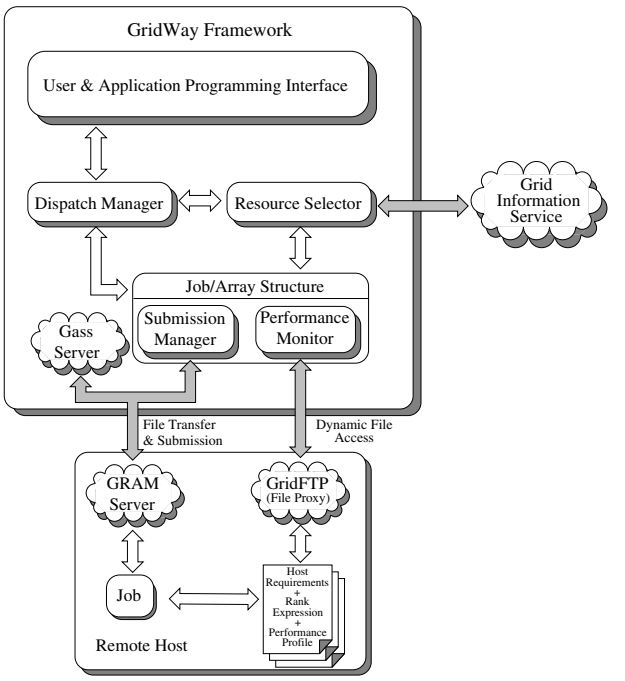

Figure 1. The Grid*W*ay architecture.

- The *resource selector* module, which is used by the *dispatch manager* to build a prioritized list of candidate resources following the preferences and requirements provided by the user (see Section 3).
- The *performance evaluator* module, which is used by the *performance monitor* to periodically evaluate the application performance.

3. Resource Selection

In order to adapt the execution of a job to its dynamic demands, the application must specify its host requirements through a `requirement expression`. The application could define an initial set of requirements and dynamically change them when more, or even less, resources are required. Also, in order to prioritize the resources that fulfil the requirements according to its runtime needs, the application must specify its hosts preferences through a `ranking expression`. A compute-intensive application would assign a higher rank to those hosts with faster CPUs and lower load, while a data-intensive application could benefit those hosts closer to the input data.

In the experiments described in the Section 7, the application does not impose any requirement to the resources. The `ranking expression` uses a performance model to estimate the job turnaround time as the sum of execution and transfer time, derived from the performance and prox-

imity of the candidate resources [7]. The application doesn't dynamically change its resource demands.

The `requirement expression` and `ranking expression` files are used by the *resource selector* to build a list of potential execution hosts. Figure 2 shows the resource selection process. Initially, available compute resources are discovered by accessing the GIIS server and those resources that do not meet the user-provided requirements are filtered out. At this step, an authorization test is performed to guarantee user access. Then, the resource is monitored by accessing its local GRIS server. The information gathered is used to assing a rank to each candidate resource based on the user-provided preferences. Finally, the resultant prioritized list of candidate resources is used to dispatch the jobs.

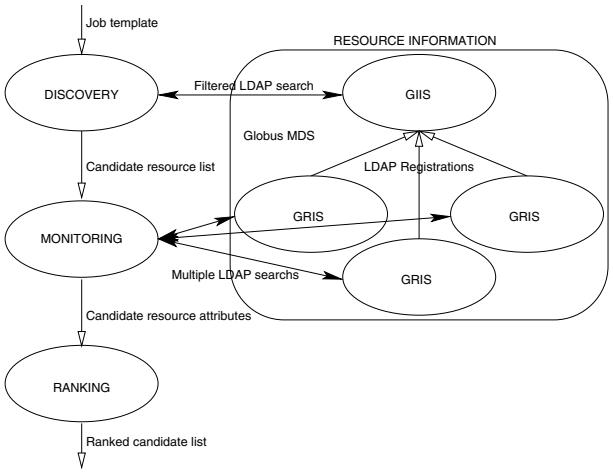

Figure 2. Resource selection process.

In order to reduce the information retrieval overhead, the GIIS and GRIS information is locally cached at the client host and updated independently in order to separately determine how often the testbed is searched for new resources and the frequency of resource monitoring. In the following experiments we set the GIIS cache timeout to 5 minutes and the GRIS cache timeout to 1 minute.

4. Job Execution

Job execution is performed in three steps by the following modules:

- The *prolog* module, which is responsible for creating the remote experiment directory and transferring the executable and all the files needed for remote execution, such as input or restart files corresponding to the execution architecture. These files can be specified as local files in the experiment directory or as remote files stored in a file server through a GridFTP URL. For the files declared as *shared*, a reference is added to the remote GASS cache, so they can be re-used by other jobs.
- The *wrapper* module, which is responsible for executing the actual job and obtaining its exit code. The capture of the remote execution exit code allow users to define complex jobs, where each depends on the output and exit code from the previous job. They may even involve branching, looping and spawning of subtasks, allowing the exploitation of the parallelism on the work flow of certain type of applications.
- The *epilog* module, which is responsible for transferring back output files, and cleaning up the remote experiment directory. At this point, references to *shared* files in the GASS cache are also removed.

Due to the high fault rate and the dynamic rescheduling, the application must generate `restart files` in order to restart the execution from a given point. If these files are not provided, the job is restarted from the beginning. User-level checkpointing managed by the programmer must be implemented because system-level checkpointing is not currently possible among heterogeneous resources. The PSA used in Section 7 has been modify to periodically generate an architecture-independent restart file.

Migration is performed by conveniently combining the above stages. The *wrapper* is canceled (if it is still running), then the *prolog* is submitted to the new candidate resource, preparing it and transferring all the needed files to it, including the `restart files` from the old resource. After that, the *epilog* is submitted to the old resource (if it is still available), but no output file staging is performed, it only cleans up the remote system. And finally, the *wrapper* is submitted to the new candidate resource.

5. Fault Tolerance Support

The *submission agent* provides the application with the fault detection capabilities needed in such a faulty environment:

- The GRAM *job manager* notifies submission failures as GRAM callbacks. This kind of failures includes connection, authentication, authorization, RSL parsing, executable or input staging, credential expiration and other failures.
- The GRAM *job manager* is probed periodically at each *polling* interval. If the *job manager* does not respond, the GRAM *gatekeeper* is probed. If the *gatekeeper* responds, a new *job manager* is started to resume watching over the job. If the *gatekeeper* fails to respond, a resource or network occurred. This is the approach followed in Condor/G [6].

- The standard output of *prolog*, *wrapper* and *epilog* is parsed in order to detect failures. In the case of the *wrapper*, this is useful to capture the job exit code, which is used to determine whether the job was successfully executed or not. If the job exit code is not set, the job was prematurely terminated, so it failed or was intentionally canceled.

When an unrecoverable failure is detected, the *submission agent* retries the submission of *prolog*, *wrapper* or *epilog* a number of times specified by the user and, when no more retries are left, it performs an action chosen by the user among two possibilities: stop the job for manually resuming it later, or automatically generate a *rescheduling* event.

6. Grid Scheduling Policy

Grid scheduling or superscheduling [5], has been defined in the literature as the process of scheduling resources over multiple administrative domains based upon a defined policy in terms of job requirements, system throughput, application performance, budget constraints, deadlines, etc. In particular previous works have demonstrated the relevance of the following factors:

1. *Adaptive Scheduling*: Reliable schedules can only be issued considering the dynamic characteristics of the available Grid resources [2, 3, 4, 1]. In general, adaptive scheduling can consider, among others, factors such as availability, performance, load, proximity (in terms of bandwidth and latency of the interconnection links), etc, properly scaled according to the application needs. Adaptive scheduling is implemented in the Grid*W*ay framework by the *dispatch manager*. The *dispatch manager* periodically gathers information from the Grid to adaptively schedule pending tasks according to the application preferences (`rank expression`) and Grid resource status, as explained in Section 3.
2. *Adaptive Execution*: In order to obtain a reasonable degree of both application performance and fault tolerance, a job must be able to migrate among the Grid resources adapting itself to events dynamically generated by both the Grid and the running application [10, 9, 14] (see Section 2). Grid*W*ay evaluates each *rescheduling* event to decide if the migration is feasible and worthwhile. Some reasons, like job cancellation or failure, make the *dispatch manager* immediately trigger a *migration* event to the *submission manager*. Other reasons, like new resource discovery, make the *dispatch manager* trigger a *migration* event only if the new selected host presents a higher enough rank. In this case, the time to finalize [14, 10], input and restart file transfer costs [7] are also considered.

Resource	Model	Speed (MHz)	Mem. (MB)	Nodes
ursa	Sun Blade 100	500	256	1
draco	Sun Ultra 1	167	128	1
pegasus	Intel Pentium 4	2400	1024	1
solea	Sun Enterp. 250	296	256	2
babieca	Alpha DS10	466	1024	4

Table 1. Summary of the UCM-CAB grid resource hardware characteristics.

Probably one of the most relevant factors when scheduling parameter sweep applications is the impact of file transfer times. In order to efficiently execute this kind of applications, the schedule should promote the re-use of shared files between tasks. Several heuristics to schedule parameter sweep applications considering file I/O requirements have been proposed in the literature [4]. The Grid*W*ay framework take advantage of file sharing by using the Globus GASS cache, and by overlapping the *prolog* and *epilog* phases (see Section 4) of two different tasks on the same resource.

Figure 3 shows the general structure of the schedule algorithm employed by the Grid*W*ay framework. This algorithm merge the previous considerations to efficiently execute parameter sweep applications on computational Grids.

```
resource selection
while (there are pending tasks and there are available slots) {      ┐ Adaptive
   submit next pending task to next available slot                   │ scheduling
}                                                                    ┘
while (there are rescheduled tasks and there are available slots) {  ┐
   evaluate migration                                                │
   if (migration granted) {                                          │ Adaptive
      migrate next rescheduled task to next available slot           │ execution
   }                                                                 │
}                                                                    ┘
```

Figure 3. Scheduling algorithm.

7. Experiences

Let us now consider a parameter sweep application consisting of 50 independent tasks. Each task calculates the flow over a flat plate [12] for a different Reynolds number, ranging from 10^2 to 10^4. In this experiment we have used the UCM-CAB testbed, whose main characteristics are summarized in tables 1 and 2.

The experiment files consists of the executable (0.5MB) and the computational mesh (0.5MB) provided for all the resource architectures in the testbed, and some parameter

Resource	OS	GRAM	VO
ursa	Solaris 8	fork	UCM
draco	Solaris 8	fork	UCM
pegasus	Linux 2.4	fork	UCM
solea	Solaris 8	fork	UCM
babieca	Linux 2.4	PBS	CAB

Table 2. Summary of the UCM-CAB grid resource software characteristics.

files (1K) describing the numerical simulation. The final file name of the executable and the computational mesh are obtained by resolving the framework variable `GW_ARCH` at runtime for the selected host. Equivalently, the final file name of the parameter file is resolved with the variable `GW_TASK_ID` for each task. In the following experiments the computational mesh and the executable are declared as *shared* files and stored in the GASS cache, so they can be re-used by different task submitted to the same resource.

Figure 4 presents the average job turnaround, execution and file transfer times on each host of the UCM-CAB Grid; error bars represent the standard deviation of these measurements. These times include the overhead induced by the Globus middleware. The standard deviation is greater for the measurements performed on babieca and solea. This fact clearly reflects the dynamicity of the Grid, outlined in Section 1. The execution times on babieca includes the queue wait time on the PBS batch system, which causes variable execution times. Also, file transfer times between the client (UCM) and babieca (CAB) exhibit a high standard deviation, due to the dynamic bandwidth of the interconnection link between these two VOs. In the case of solea, the high standard deviation is due to a non-exclusive access to the system, mainly used as web and mail server.

The overall execution time for parameter sweep application, when all the machines in the testbed are available, is 2310 seconds with an average job turnaround time of 46.2 seconds. Compared to the single host execution on the fastest machine in the testbed (pegasus, 62 seconds per job), these results represents 25% reduction in the overall execution time. This experiment is repeated introducing an artificial workload on pegasus at the middle of the execution. As could be expected, the workload on pegasus increases the dynamic and average (52.4 seconds) job turnaround times, as well as the overall execution time (2622 seconds). Figure 5 shows the dynamic job turnaround time during the execution of the parameter sweep application in the above situations.

We will next evaluate the schedule performed by the Grid*Way* framework in the above experiments compared to the optimum *static* schedule. This comparison can only be

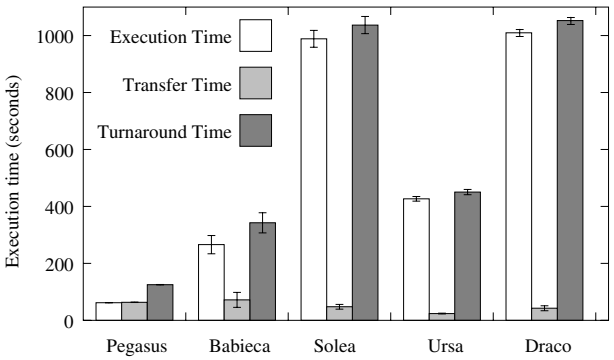

Figure 4. Average and standard deviation in turnaround time, file transfer time and execution time for the parameter sweep application on each host of the testbed.

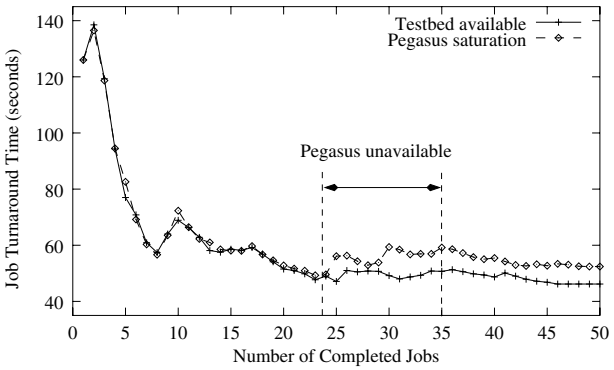

Figure 5. Dynamic job turnaround time in the execution of the parameter sweep application when the testbed is fully available and when pegasus is saturated.

seen as a reference, whose main goals are to establish an upper performance bound and, to highlight the relevance of adaptive scheduling in a Grid environment. The optimum Grid schedule will minimize the makespan of the application [4]:

$$\begin{aligned} &\textbf{minimize} \\ &\quad \max\{N_i\overline{T}_i\}\ \forall i \in GR \\ &\textbf{subject to} \\ &\quad \sum_{i \in GR} N_i - 50 = 0; \\ &\quad 0 \leq N_i \leq 50\ \forall i \in GR, \end{aligned} \tag{1}$$

where N_i is the number of jobs executed on host i, $\overline{T}_i$ is the average job turnaround time on host i, and GR is the set of all Grid resources ($GR = \{pegasus,\ draco,\ solea,\ ursa,\ babieca\}$).

Figure 6 shows the number of jobs scheduled on each host by the Grid*W*ay framework, and the solution to the optimization problem 1. The overall execution time for the optimum schedule is 2105 seconds, with an average job turnaround time of 42.1 seconds. The optimum schedule is 9% better than the schedule made by the Grid*W*ay framework when the testbed is fully available.

The main difference between the schedules made by the Grid*W*ay framework when the testbed is fully available and when pegasus is saturated, is the greater number of jobs allocated to pegasus. Since the Grid*W*ay framework detects the saturation of pegasus and dynamically schedules pending jobs to other hosts. In this case, the adaptive schedule made by the Grid*W*ay framework only increments the execution time 13%, although pegasus was unavailable for 970 seconds, 37% of the experiment.

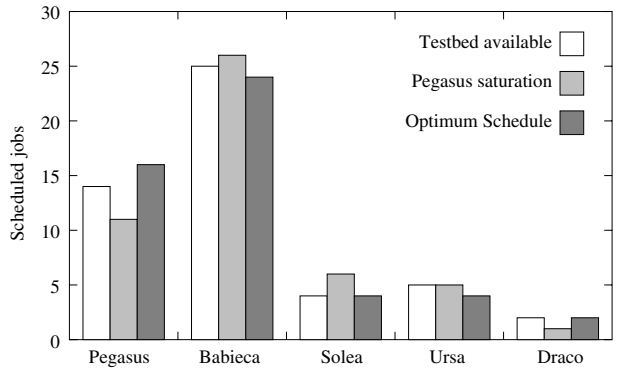

Figure 6. Number of jobs scheduled on each resource by the optimum schedule and by the Grid*W*ay framework when the testbed is fully available and when pegasus is saturated.

8. Conclusions

In this paper we have extended the capabilities of the Grid*W*ay framework to enhanced the performance of PSAs. The Grid*W*ay framework achieves the robust and efficient execution of PSAs by combining: adaptive scheduling to reflect the dynamic Grid characteristics, adaptive execution to migrate running jobs to *better* resources and provide fault tolerance, and re-use of common files between tasks to reduce the file transfer overhead. The experimental results are promising because they show how generic PSAs can efficiently harness the highly distributed computing resources provided by a Grid. Currently, the ideas and middleware developed in this work are being used in the execution of an existing bioinformatics application to study large numbers of protein structures.

References

[1] G. Allen et al. The Cactus Worm: Experiments with Dynamic Resource Discovery and Allocation in a Grid Environment. *Intl. J. of High-Performance Computing Applications*, 15(4), 2001.

[2] F. Berman et al. Adaptive Computing on the Grid Using AppLeS. *IEEE Transactions on Parallel and Distributed Systems*, 14(5):369–382, 2003.

[3] R. Buyya et al. Nimrod/G: An Architecture for a Resource Management and Scheduling System in a Global Computation Grid. In *Proc. of the 4th Intl. Conf. on High Performance Computing in Asia-Pacific Region (HPC Asia)*, 2000.

[4] H. Casanova et al. Heuristics for Scheduling Parameter Sweep Applications in Grid Environments. In *Proc. of the 9th Heterogeneous Computing Workshop*, 2000.

[5] I. Foster and C. Kesselman. *The Grid: Blueprint for a New Computing Infrastructure*. Morgan-Kaufman, 1999.

[6] J. Frey et al. Condor/G: A Computation Management Agent for Multi-Institutional Grids. In *Proc. of the 10th Symp. on High Performance Distributed Computing (HPDC10)*, 2001.

[7] E. Huedo, R. S. Montero, and I. M. Llorente. Experiences on Grid Resource Selection Considering Resource Proximity. In *Proc. of 1st European Across Grids Conf.*, 2003.

[8] E. Huedo, R. S. Montero, and I. M. Llorente. A Framework for Adaptive Execution on Grids. *Intl. J. of Software – Practice and Experience*, 2003. To appear.

[9] G. Lanfermann et al. Nomadic Migration: A New Tool for Dynamic Grid Computing. In *Proc. of the 10th Symp. on High Performance Distributed Computing (HPDC10)*, 2001.

[10] R. S. Montero, E. Huedo, and I. M. Llorente. Experiences about Job Migration on a Dynamic Grid Environment. In *Proc. of Intl. Conf. on Parallel Computing (ParCo 2003)*, Advances on Parallel Computing. Elsevier Science, September 2003. To appear.

[11] R. S. Montero, E. Huedo, and I. M. Llorente. Grid Resource Selection for Opportunistic Job Migration. In *Proc. of Intl. Conf. on Parallel and Distributed Computing (Euro-Par 2003)*, LNCS. Springer-Verlag, August 2003.

[12] R. S. Montero, I. M. Llorente, and M. D. Salas. Robust Multigrid Algorithms for the Navier-Stokes Equations. *Journal of Computational Physics*, 173:412–432, 2001.

[13] J. M. Schopf. Ten Actions when Superscheduling. Technical Report WD8.5, The Global Grid Forum, Scheduling Working Group, 2001.

[14] S. Vadhiyar and J. Dongarra. A Performance Oriented Migration Framework for the Grid. In *Proc. of the 3rd Intl. Symp. on Cluster Computing and the Grid (CCGrid)*, 2003.