Grid Resource Selection for Opportunistic Job Migration *

Rubén S. Montero[1], Eduardo Huedo[2], and Ignacio M. Llorente[1,2]

[1] Departamento de Arquitectura de Computadores y Automática, Universidad Complutense, 28040 Madrid, Spain.
[2] Centro de Astrobiología (Associated to NASA Astrobiology Institute), CSIC-INTA, 28850 Torrejón de Ardoz, Spain.

Abstract. The ability to migrate running applications among different grid resources is generally accepted as the solution to provide fault tolerance and to adapt to dynamic resource load, availability and cost. In this paper we focus on opportunistic migration when a new resource becomes available in the grid. In this situation the performance of the new host, the remaining execution time of the application, and also the proximity of the new resource to the needed data, become critical factors to decide if job migration is feasible and worthwhile. We discuss the extension of the GridWay framework to consider all the previous factors in the resource selection and migration stages in order to improve response times of individual applications. The benefits of the new resource selector will be demonstrated for the execution of a computational fluid dynamics (CFD) code.

1 Introduction

Computational Grids are inherently dynamic environments, being characterized by unpredictable changing conditions, namely:

- *High fault rate*: In a Grid, resource or network failures are the rule rather than the exception.
- *Dynamic resource availability*: Grid resources belong to different administrative domains; so that, once a job is submitted, it can be freely canceled by the resource owner. Furthermore, the resources shared within a virtual organization can be added or removed continuously.
- *Dynamic resource load*: Grid users access resources that are being exploited by other grid users, as well as by internal users. This fact may cause that initially idle hosts become saturated, and vice versa.
- *Dynamic resource cost*: In an economy driven grid [1], resource prices can vary depending on the time of the day (working/non-working time) or the resource load (peak/off-peak).

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

Consequently, 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 according to their availability, load, and cost.

Probably, the most sensitive step to the above conditions in job scheduling is resource selection, which in turn relies completely in the dynamic information gathered from the grid. Resource selection usually takes into account the performance offered by the available resources, but it should also consider the proximity between them [2, 3]. The size of the files involved in some application domains, like Particle Physics or Bioinformatics, is very large. Hence, the quality of the interconnection between resources, in terms of bandwidth and latency, is a key factor to be considered in resource selection [4]. This fact is specially relevant in the case of adaptive job execution, since job migration requires the transfer of large restart files between the compute hosts. In this case, the quality of the interconnection network has a decisive impact on the overhead induced by job migration.

In this paper, we focus on the opportunistic migration of jobs when a new "better" resource is discovered, because either a new resource is added to the grid, or because the completion of an application frees a grid resource. Opportunistic migration has been widely studied in the literature [5–8], previous works have clearly demonstrated the relevance of considering the amount of the computational work already performed by the application, the need of a metric to measure the performance gain due to migration, and the critical factor of dynamic load information of grid resources. However, previous migration frameworks do not consider the proximity of the computational resources to the needed data, and therefore the potential performance gain can be substantially decremented by the cost of data transfer.

The migration and brokering strategies presented in this work have been implemented on top of the GridWay framework [9], whose architecture and main functionalities are briefly described in section 2. In section 3 we discuss the extension of the GridWay framework to also consider resource proximity in the resource selection stage. This selection process is then incorporated to the GridWay migration system in section 4. The benefits of the new resource selector will be demonstrated in section 5 for the adaptive execution of a computational fluid dynamics (CFD) code on a research testbed. Finally, section 6 includes some conclusions and outlines our future work.

2 The GridWay Framework

GridWay is a new 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 GridWay framework has been designed to strictly meet the following guidelines:

- easily adaptable, through a modular and flexible design.
- easily scalable, thanks to its decentralized architecture, although some of the grid services used, mainly the information services, can be centralized.

- easily and widely deployable, since it executes and installs as a user program and works with the available grid services.
- easily extensible, since it can use other non-standard services.
- easily and widely applicable, as it is ready to use on a dynamic and faulty environment for a wide range of applications.

The core of the GridWay framework is a personal *submission agent* that automatically performs the steps involved in job submission: system selection, system preparation, submission, monitoring, migration and termination [10]. 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 [9] for a detailed description of these components).

The flexibility of the framework is guaranteed by a well-defined API (Application Program Interface) for each *submission agent* component. Moreover, the framework has been designed to be modular, through scripting, to allow extensibility and improvement of its capabilities. The following modules can be set on a per job basis:

- *resource selector*, which acts as a personal resource broker to build a prioritized list of candidate resources following the preferences and requirements provided by the user.
- *performance degradation evaluator*, which is used to periodically evaluate the application performance.
- *prolog*, which prepares the remote system and performs executable and input file staging.
- *wrapper*, which executes the job and returns its exit code.
- *epilog*, which performs output file staging and cleans up the remote system.

3 The Resource Selector

Due to the heterogeneous and dynamic nature of the grid, the end-user must establish the requirements that must be met by the target resources (discovery process) and criteria to rank the matched resources (selection process). The attributes needed for resource discovery and selection must be collected from the information services in the grid testbed, typically the Globus Monitoring and Discovery Service (MDS). Usually, resource discovery is only based on static attributes (operating system, architecture, memory size...) collected from the Grid Information Index Service (GIIS), while resource selection is based on dynamic attributes (disk space, processor load, free memory...) obtained from the Grid Resource Information Service (GRIS).

The dynamic network bandwidth and latency between resources will be also considered in the resource brokering scheme. Different strategies to obtain these

network performance attributes can be adopted depending on the services available in the testbed. For example, MDS could be configured to provide such information by accessing the Network Weather Service (NWS) [11] or by activating the reporting of GridFTP statistics [12]. Alternatively, the end-user could provide its own network probe scripts or static tables.

The brokering process of the GridWay framework is shown in figure 1. 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 (via GRAM ping request) is also performed on each discovered host to guarantee user access to the remote resource. Then, the dynamic attributes of each host are gathered from its local GRIS server. This information is used by an user-provided rank expression to assign a rank to each candidate resource. Finally, the resultant prioritized list of candidate resources is used to dispatch the job.

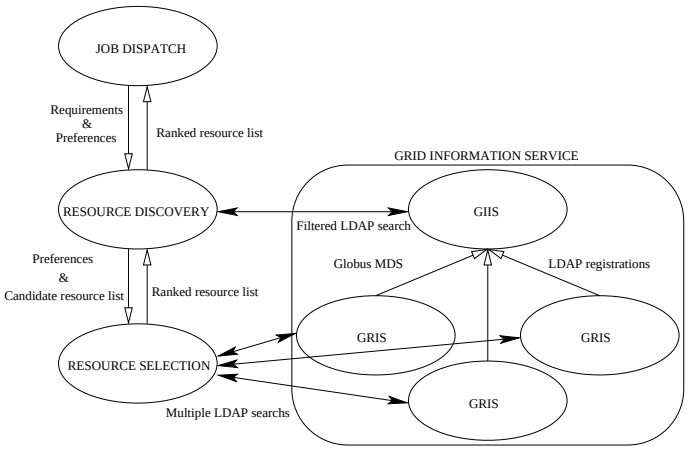

Fig. 1. The brokering process scheme of the GridWay framework.

The resource selection overhead is determined by the cost of retrieving the dynamic and static resource information, and the scheduling process itself. In the present case the cost of scheduling jobs, i.e. rank calculation, can be neglected compared to the cost of accessing the MDS, which can be extremely high [13]. In order to reduce the information retrieval overhead, the GIIS information is locally cached at the client host and updated every hour, this update frequency determines how often the testbed is searched for new resources. The GRIS contents are also cached locally but updated every minute, since the CPU availability information, used in rank calculation (see section 3.1), is generated every minute by the GRIS provider.

The new selection process presented in this paper considers both dynamic performance and proximity to data of the computational resources. In particular, the following circumstances will be considered in the resource selection stage:

- The estimated computational time on the candidate host being evaluated when the job is submitted from the *client* or migrated from the current or last *execution host*.
- The proximity between the candidate host being evaluated and the *client* will be considered to reduce the cost of job submission, job monitoring and file staging.
- The proximity between the candidate host being evaluated and a remote *file server* will be also considered to reduce the transfer costs when some input or output files, specified as a GridFTP URL, are stored in such server.
- The proximity between the candidate host being evaluated and the current or last *execution host* will be also considered to reduce the migration overhead, since the transfer of restart files is performed between them.

3.1 Performance Model

In order to reflect all the circumstances described previously, each candidate host (h_n) will be ranked using the *total* execution time (lowest is best) when the job is submitted or migrated to that host at a given time (t_n). In this case, we can assume that the total execution time can be split into:

$$T_{exe}(h_n, t_n) = T_{cpu}(h_n, t_n) + T_{xfer}(h_n, t_n), \tag{1}$$

where $T_{cpu}(h_n, t_n)$ is the estimated computational time and $T_{xfer}(h_n, t_n)$ is the estimated file transfer time.

Let us first consider a single-host execution, the computational time of the application on host h at time t can be estimated by:

$$T^s_{cpu}(h,t) = \begin{cases} \frac{Op}{FLOPS} & \text{if } CPU(t) \geq 1; \\ \frac{Op}{FLOPS \cdot CPU(t)} & \text{if } CPU(t) < 1. \end{cases} \tag{2}$$

where $FLOPS$ is the peak performance achievable by the host CPU, Op is the number of floating point operations of the application, and $CPU(t)$ is the total free CPU at time t, as provided by the MDS default scheme.

However, the above expression is not accurate when the job has been executing on multiple hosts and then is migrated to a new one. In this situation the amount of computational work that have already been performed must be considered [6]. Let us suppose an application that has been executing on hosts $h_0 \ldots h_{n-1}$ at times $t_0 \ldots t_{n-1}$ and then migrates to host h_n at time t_n, the overall computational time can be estimated by:

$$T_{cpu}(h_n, t_n) = \sum_{i=0}^{n-1} t^i_{cpu} + \left(1 - \sum_{i=0}^{n-1} \frac{t^i_{cpu}}{T^s_{cpu}(h_i, t_i)}\right) T^s_{cpu}(h_n, t_n), \tag{3}$$

where $T^s_{cpu}(h,t)$ is calculated using (2), and t^i_{cpu} is the time the job has been executing on host h_i, as measured by the framework. Note that, expressions 2 and 3 become equivalent when $n = 0$.

Similarly, the following expression estimates the total file transfer time:

$$T_{xfer}(h_n, t_n) = \sum_{i=0}^{n-1} t^i_{xfer} + \sum_j \frac{Data_{h_n,j}}{bw(h_n, j, t_n)} \quad j = client, file\ server, exec\ host, \tag{4}$$

where $bw(h_1, h_2, t)$ is the bandwidth between hosts h_1 and h_2 at time t, $Data_{h_1,h_2}$ is the file size to be transferred between them, and t^i_{xfer} is the file transfer time on host h_i, as measured by the framework.

4 GridWay Support for Adaptive Job Execution

The GridWay framework supports job adaption to changing conditions by means of automatic job migration. Once the job is initially allocated, it is dynamically rescheduled when one of the following events occurs:

- a new "better" resource is discovered (opportunistic migration),
- the remote host or its network connection fails,
- the submitted job is canceled or suspended by the remote resource administrator,
- a performance degradation is detected,
- the requirements or preferences of the application changes (self-migration).

In this work we will concentrate on opportunistic migration. The *dispatch manager* wakes up at each discovery interval, and it tries to find a better host for each job by activating the resource selection process described in section 3. In order to evaluate the benefits of job migration from the current execution host (h_{n-1}) to each candidate host (h_n), we define the migration gain (G_m) as:

$$G_m = \frac{T_{exe}(h_{n-1}, t_{n-1}) - T_{exe}(h_n, t_n)}{T_{exe}(h_{n-1}, t_{n-1})}, \tag{5}$$

where $T_{exe}(h_{n-1}, t_{n-1})$ is the estimated execution time on current host when the job was submitted to that host, and $T_{exe}(h_n, t_n)$ is the estimated execution time when the application is migrated to the new candidate host. The migration is granted only if the migration gain is greater than an user-defined threshold, otherwise it is rejected. Note that although the migration threshold is fixed for a given job, the migration gain is dynamically computed to take into account the dynamic data transfer overhead, the dynamic host performance, and the application progress. In the experiments presented in section 5 the migration gain has been fixed to 10%.

Migration is implemented by restarting the job on the new candidate host: first the job is canceled, and then the checkpoint and standard output files are transferred from the current execution host. In the current version of our

framework, users must explicitly manage their own checkpoint data, which must be architecture independent (ASCII or HDF), in order not to restrict the range of feasible candidates. We expect in future versions to incorporate the Grid Checkpoint Recovery Application Programming Interface under specification by the Grid Checkpoint Recovery Working Group of the Global Grid Forum.

5 Experiments

The behavior of the resource selection strategy previously described is demonstrated in the execution of a CFD code, that solves the 3D incompressible Navier-Stokes equations using an iterative multigrid method [14]. In the following experiments, the *client* host is ursa, which holds an input file with the simulation parameters, and the *file server* is cepheus, which holds the executable and the computational mesh. The output file with the velocity and pressure fields is transferred back to the *client*, ursa, to perform post-processing. Table 1 shows the available machines in the testbed, their corresponding CPU performance (MFLOPS), and the maximum bandwidth (MB/s) between them and the hosts involved in the experiment.

Table 1. Available machines in the testbed, their CPU performance, and bandwidth between them and the machines involved in the experiment (*client*=ursa, *file server*=cepheus and *exec host*=draco).

					Bandwidth		
host	Model	*CPU*	OS	Memory	*client*	*file server*	*exec host*
ursa	Sun Blade 100	330	Solaris 8	256MB	∞	0.4	0.4
draco	Sun Ultra 1	175	Solaris 8	128MB	0.4	0.4	∞
columba	Pentium MMX	225	Linux 2.4	160MB	0.4	0.4	0.4
cepheus	Pentium Pro	325	Linux 2.4	64MB	0.4	∞	0.4
solea	Sun Enterprise 250	350	Solaris 8	256MB	0.2	0.2	0.2

We will impose two requirements on the compute resources: a minimum main memory of 128MB, enough to accommodate the CFD simulation; and a total free CPU higher than 90% to prevent oscillating migrations. Initially, the application is submitted to draco, since it is the only resource that meet the previous requirements. We will evaluate re-scheduling strategies based on expressions 3 and 1, when the artificial workload running on columba and solea completes at different execution points (iterations) of the application running on draco.

Let us first suppose that the application is re-scheduled using expression 3 (figure 2, left-hand chart). In this case, as the file transfer time is not considered, migration to both hosts always presents a performance gain. The *dispatch manager* will consider feasible the migration to the lowest ranked host, solea, until the eight iteration ($T_{exe} = 302$) is reached.

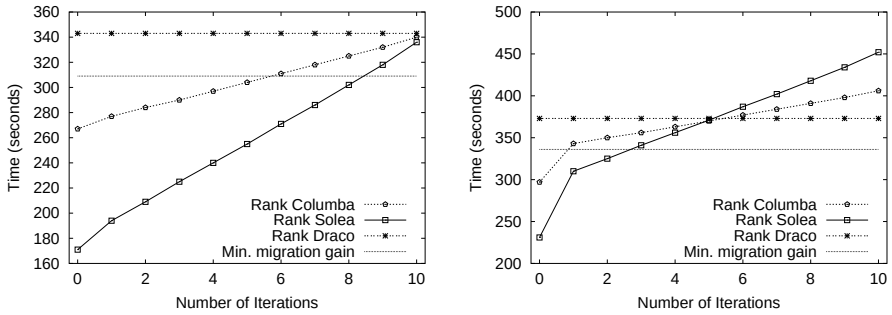

Fig. 2. Estimated execution times (ranks) of the application when it is migrated from darco to different machines at different execution points, using expressions 3 (left-hand chart) and 1 (right-hand chart).

Figure 2 (right-hand chart) shows the dynamic ranks of solea and columba when the application is re-scheduled using expression 1. In this situation, migration to solea only will be granted until the second iteration ($T_{exe} = 325$) is reached. Note that from fifth iteration, the performance gain offered by solea and columba is not high enough to compensate the file transfer overhead induced by job migration. Moreover, from sixth iteration the lowest ranked host is columba (nearest host) although it presents a CPU performance lower than solea, as proximity to data becomes more important as the application progresses.

Figure 3 shows the measured execution profile of the application when it is actually migrated to solea and columba at different iterations, and the execution profile on draco without migration. These experimental results clearly show that re-schedules based only on host performance and application progress (expression 3) may not yield in performance benefits. In particular, re-scheduling the job based on expression 1 results in a performance gain of 13% (12% predicted). This resource selection strategy will reject job migration from third iteration, and prevents performance loses up to 15%, that would occur with the re-scheduling strategy based on expression 3. Note that this drop in performance can always be avoided with a pessimistic value of G_m. However using expression 1 allows a more aggressive value of the migration gain threshold, and therefore an improvement in the response time of the application.

6 Conclusions and Future Work

In this work we have analyzed the relevance of resource proximity in the resource selection process, in order to reduce the cost of file staging. In the case of opportunistic migration the quality of the interconnection network has also a decisive impact on the overhead induced by job migration. In this way, considering resource proximity to the needed data is, at least, as important as considering resource performance characteristics. We expect that resource proximity would be even more relevant for greater file sizes and more heterogeneous networks.

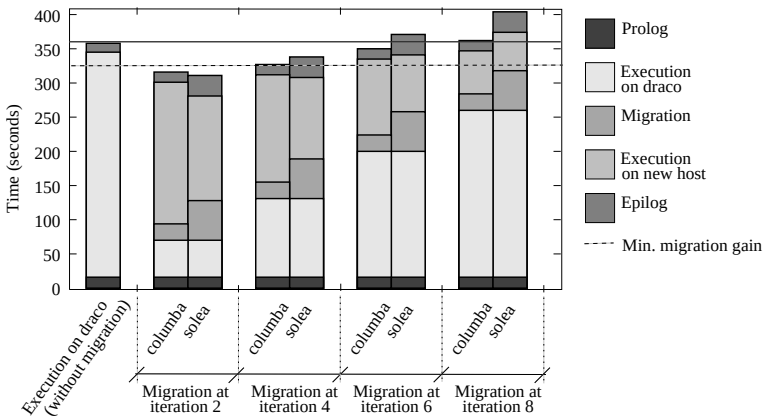

Fig. 3. Execution profile of the application when it is executed on draco (left bar in the chart), and when it is migrated from draco to solea or columba at different execution points.

We would like to note that the decentralized and modular architecture of the GridWay framework guarantees the scalability of the brokering strategy, as well as the range of application, since it is not specialized for a specific application set.

We are currently applying the same ideas presented here to develop a *storage resource selector* program that considers the proximity to a set of replica files belonging to a logical collection. The storage resource selection process is equivalent to the one presented in figure 1, although the discovery process is performed by accessing the Globus Replica Catalog. The resource selection is based on the bandwidth between the selected compute resource and the candidate storage resources, along with the values gathered from the MDS GRIS.

References

1. Buyya, R., D.Abramson, Giddy, J.: A Computational Economy for Grid Computing and its Implementation in the Nimrod-G Resource Broker. Future Generation Computer Systems (2002) Elsevier Science (to appear).
2. Liu, C., Yang, L., Foster, I., Angulo, D.: Design and Evaluation of a Resource Selection Framework for Grid Applications. In: Proceedings of the 11th IEEE Symposium on High-Performance Distributed Computing. (2002)
3. Kennedy, K., Mazina, M., Mellor-Crummey, J., Cooper, K., Torezon, L., Berman, F., Chien, A., Dail, H., Sievert, O., Angulo, D., Foster, I., Gannon, D., Johnsson, L., C-Kesselman, Aydt, R., Reed, D.A., Dongarra, J., Vadhiyar, S., Wolski, R.: Toward a Framework for Preparing and Execution Adaptive Grid Applications. In: Proceedings of NSF Next Generation Systems Program Workshop, International Parallel and Distributed Processing Symposium. (2002) Fort Lauderdale, USA.

4. Allcock, W., Chervenak, A., Foster, I., Pearlman, L., Welch, V., Wilde, M.: Globus Toolkit Support for Distributed Data-Intensive Science. In: Proceedings of Computing in High Energy Physics (CHEP '01). (2001)
5. Evers, X., de Jongh, J.F.C.M., Boontje, R., Epema, D.H.J., van Dantzig, R.: Condor Flocking: Load Sharing Between Pools of Workstations. Technical Report DUT-TWI-93-104, Delft, The Netherlands (1993)
6. Vadhiyar, S., Dongarra, J.: A Performance Oriented Migration Framework for the Grid. In: Proceedings of the 3rd IEEE/ACM Int'l Symposium on Cluster Computing and the Grid (CCGrid). (2003)
7. Wolski, R., Shao, G., Berman, F.: Predicting the Cost of Redistribution in Schedulling. In: Proceedings of the 8th SIAM Conference on Parallel Processing for Scientific Applications. (1997)
8. Allen, G., et al.: The Cactus Worm: Experiments with Dynamic Resource Discovery and Allocation in a Grid Environment. International Journal of High-Performance Computing Applications **15** (2001)
9. Huedo, E., Montero, R.S., Llorente, I.M.: An Experimental Framework for Executing Applications in Dynamic Grid Environments. Technical Report 2002-43, ICASE NASA Langley (2002) submitted to Software Practice & Experience Journal.
10. Schopf, J.M.: A General Architecture for Scheduling on the Grid. Available at http://www-unix.mcs.anl.gov/~schopf (2002) Submitted to special issue of Journal of Parallel and Distributed Computing on Grid Computing.
11. Wolski, R., Spring, N., Hayes, J.: The Network Weather Service: A Distributed Resource Performance Forecasting Service for Metacomputing. Journal of Future Generation Computing Systems **15** (1999) 757–768
12. Vazhkudai, S., Schopf, J., Foster, I.: Predicting the Performance of Wide-Area Data Transfers. In: Proceedings of 16th Int'l Parallel and Distributed Processing Symposium (IPDPS 2002). (2002)
13. Dail, H., Casanova, H., Berman, F.: A Decoupled Scheduling Approach for the GrADS Program Development Enviroment. In: Proceedings of the SuperComputing (SC02). (2002) Maryland, USA.
14. Montero, R.S., Llorente, I.M., Salas, M.D.: Robust Multigrid Algorithms for the Navier-Stokes Equations. Journal of Computational Physics **173** (2001) 412–432