Scalable Computing: Practice and Experience
Volume 6, Number 3, pp. 1–8. http://www.scpe.org

ISSN 1895-1767
© 2005 SWPS

THE GRIDWAY FRAMEWORK FOR ADAPTIVE SCHEDULING AND EXECUTION ON GRIDS[*]

EDUARDO HUEDO[†], RUBÉN S. MONTERO[‡], AND IGNACIO M. LLORENTE[§]

Abstract.
Many research and engineering fields, like Bioinformatics or Particle Physics, are confident about the development of Grid technologies to provide the huge amounts of computational and storage resources they require. Although several projects are working on creating a reliable infrastructure consisting of persistent resources and services, the truth is that the Grid will be a more and more dynamic entity as it grows. In this paper, we present a new tool that hides the complexity and dynamicity of the Grid from developers and users, allowing the resolution of large computational experiments in a Grid environment by adapting the scheduling and execution of jobs to the changing Grid conditions and application dynamic demands.

Key words. grid technology, bioinformatics, adaptive scheduling, adaptive execution.

1. Introduction. Grid environments inherently present the following characteristics [6]: multiple administration domains, heterogeneity, scalability, and dynamicity or adaptability. These characteristics completely determine the way scheduling and execution on Grids have to be done. For example, scalability and multiple administration domains prevent the deployment of centralized resource brokers, with total control over client requests and resource status. On the other hand, the dynamic resource characteristics in terms of availability, capacity and cost, make essential the ability to adapt job execution to these conditions.

Moreover, the emerging of Grid technology has led to a new generation of applications that relies on its own ability to adapt its execution to changing conditions [5]. These new self-adapting applications take decisions about resource selection as their execution evolves, and provide their own performance activity to detect performance slowdown. Therefore self-adapting applications can guide their own scheduling.

To deal with the dynamicity of the Grid and the adaptability of the applications two techniques has been proposed in the literature, namely:

1. *Adaptive scheduling*, to allocate pending jobs to grid resources considering the available resources, their current status, and the already submitted jobs.
2. *Adaptive execution*, to migrate running jobs to more suitable resources based on events dynamically generated by both the Grid and the application.

The AppLeS [9] project has previously dealt with the concept of *adaptive scheduling*. AppLeS is currently focused on defining templates for characteristic applications, like APST for parameter sweep and AMWAT for master/worker applications. Also, the Nimrod/G [10] resource broker dynamically optimizes the schedule to meet user-defined deadline and budget constraints. On the other hand, the need of a nomadic migration [14] approach for *adaptive execution* on a Grid environment has been previously discussed in the context of the GrADS [8] project.

In the following sections, we first explain the need for an adaptive scheduling and execution of jobs due to the dynamicity of both the Grid and the application demands. Then, in Section 3, we show a Grid-aware application model. In Section 4, we present how the Grid*W*ay framework provides support for adaptive scheduling and execution. In Section 5, we show some results obtained in the UCM-CAB research testbed with a Bioinformatics application. Finally, in Section 6, we provide some conclusions and hints about our future work.

2. Adaptive Scheduling and Execution. Grid scheduling or superscheduling [11], 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,

[*] This research was supported by Ministerio de Ciencia y Tecnología (research grant TIC 2003-01321) and Instituto Nacional de Técnica Aeroespacial (INTA).

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

[‡] Departamento de Arquitectura de Computadores y Automática, Universidad Complutense, 28040 Madrid, Spain (rubensm@dacya.ucm.es).

[§] Departamento de Arquitectura de Computadores y Automática, Universidad Complutense, 28040 Madrid, Spain (llorente@dacya.ucm.es) & Laboratorio de Computación Avanzada, Centro de Astrobiología (CSIC-INTA), 28850 Torrejón de Ardoz, Spain (martinli@inta.es).

etc. In general, this process includes the following phases: resource discovery and selection; and job preparation, submission, monitoring, migration and termination [18].

Adaptive scheduling is the first step to deal with the dynamicity of the Grid. The schedule is re-evaluated periodically based on the available resources and their current characteristics, pending jobs, running jobs and history profile of completed jobs. Several projects [9, 10] have clearly demonstrated that periodic re-evaluation of the schedule in order to adapt it to the changing conditions, can result in significant improvements in both performance and fault tolerance.

In the case of adaptive execution, job migration is the key issue [15]. 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 the resource availability, load (or capacity) and cost; and to the application dynamic demands.

Consequently, the following migration circumstances, related to the changing conditions and self-adapting features both discussed in Section 1, should be considered in a Grid environment:

1. *Grid-initiated migration:*
   - A "better" resource is discovered (opportunistic migration [16]).
   - The remote resource or its network connection fails (failover migration).
   - The submitted job is canceled or suspended.
2. *Application-initiated migration:*
   - Performance degradation or performance contract violation is detected in terms of application intrinsic metrics.
   - The resource demands of the application change (self-migration).

The fundamental aspect of adaptive execution is the recognition of changing conditions of both Grid resources and application demands. In order to achieve such functionality, we propose a Grid-aware application model, which includes self-adapting functionality, and a *submission agent* that provides the runtime mechanisms needed to adapt the execution of the application. The application must be equipped with the functionality needed to support the application-initiated migration circumstances, while the agent is continuously watching the occurrence of the Grid- and application-initiated migration circumstances.

3. Application Model for Self-Adapting Applications. The standard application model requires modifications to be Grid-aware. In the following list (see figure 3.1) we detail the extension of the classical application paradigm in order to take advantage of the Grid capabilities and to be aware of its dynamic conditions:

- A `requirement expression` is necessary to specify the application requirements that must be met by the target resources. This file can be subsequently updated by the application to adapt its execution to its dynamic demands. The application could define an initial set of requirements and dynamically change them when more, or even less, resources are required.
- A `ranking expression` is necessary to dynamically assign a rank to each resource, in order to prioritize the resources that fulfill the requirements according to the application runtime needs. A compute-intensive application would assign a higher rank to those hosts with faster processors and lower load, while a data-intensive application could benefit those hosts closer to the input data [16].
- A `performance profile` is advisable to keep the application performance activity in terms of application intrinsic metrics, in order to detect performance slowdown. For example, it could maintain the time consumed by the code in the execution of a set of given fragments, in each cycle of an iterative method or in a set of given input/output operations.

Due to the high fault rate and the dynamic rescheduling, `restart files` are highly advisable. Migration is commonly implemented by restarting the job on the new candidate host, so the job should generate restart files at regular intervals in order to restart execution from a given point. However, for some application domains the cost of generating and transferring restart files could be greater than the saving in compute time due to checkpointing. Hence, if the checkpointing files are not provided the job should be restarted from the beginning. User-level checkpointing managed by the programmer must be implemented because system-level checkpointing is not possible among heterogeneous resources.

The application source code does not have to be modified if the application is not required to be self-adaptive. However, our infrastructure requires changing the source code or inserting instrumentation instructions in compiled code when the application takes decisions about resource selection and provides its own performance activity.

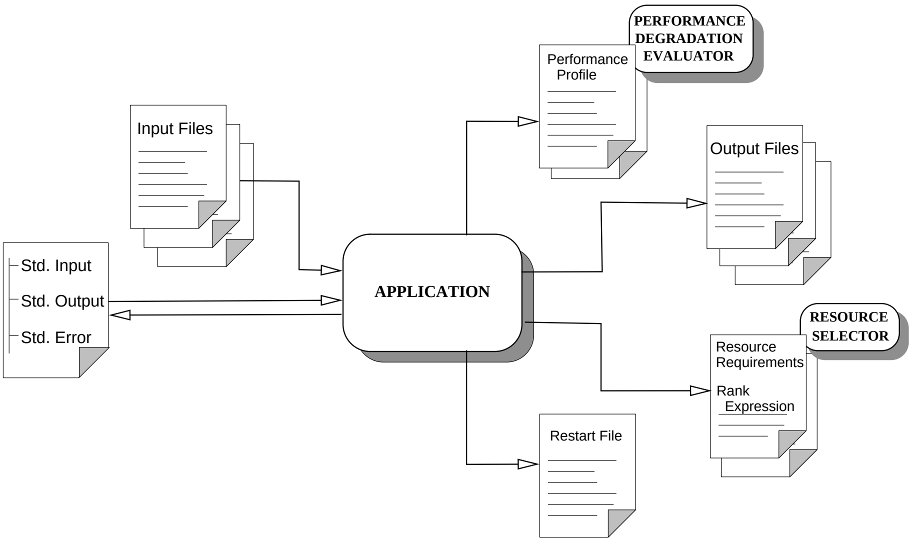

Fig. 3.1. *Model for self-adapting applications.*

With self-adapting capabilities, an application could initially define a minimal set of requirements and, after it begins to run, it can change them to a more restricted set. In this way, the application will have more chances to find a resource to run on, and once running, it will migrate only if the candidate resource worths it.

Note also that if the application is divided in several phases, each one with different requirements, it could change them progressively to be more or less restrictive. In this way, the application does not have to impose the most restricted set of requirements at the beginning, since it limits the chance for the application to begin execution (see Section 5.3.2). Moreover, the application have the choice to make the requirement change optional or mandatory, i.e. it can check if the current resource meets the new requirements, otherwise it may request a (self-)migration.

4. Grid*W*ay Support for Adaptive Scheduling and Execution. Grid*W*ay is a new experimental framework based on Globus [4] 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 [13] is a personal *submission agent* that performs all the scheduling stages [18] and watches over the correct and efficient execution of jobs. Adaptation to changing conditions is achieved by dynamic rescheduling: once the job is initially allocated, it is rescheduled when a migration circumstance (discussed in Section 2) is detected.

Job execution is performed in three stages by the following modules, which can be defined on a per job basis:

- The *prolog* module, which prepares the remote system and stages the input files.
- The *wrapper* module, which executes the actual job and returns its exit code.
- The *epilog* module, which stages the output files and cleans up the remote system.

Migration is performed by combining the above stages. First, the *wrapper* is canceled (if it is still running), then the *prolog* is submitted to the new candidate resource, preparing it and transferring to it all the needed files, 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. Finally, the *wrapper* is submitted to the new candidate resource.

The *submission agent* uses the following modules, which also can be defined on a per job basis, to provide the application with the support needed for implementing self-adapting functionality:

- The *resource selector* module, which evaluates the `requirement` and `ranking expressions` when the job has to be scheduled or rescheduled. Different strategies for resource selection can be implemented, from the simplest one based on a pre-defined list of hosts to more advanced strategies based on requirement filtering, and resource ranking in terms of performance models.
- The *performance evaluator* module, which periodically evaluates the application's `performance profile` in order to detect performance slowdown and so request a rescheduling action.

Different strategies could be implemented, from the simplest one based on querying the Grid information services about system status information to more advanced strategies based on detection of performance contract violations.

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

- The GRAM [1] *job manager* notifies submission failures as GRAM callbacks. This kind of failures includes, among others, connection, authentication, authorization, RSL parsing, executable or input staging, credential expiration...
- The *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 by Condor-G [12].
- 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 reschedule it.

We have developed both an API (subset of the DRMAA [17] standard proposed in the GGF [3]) and a command line interface to interact with the *submission agent*. They allow scientists and engineers to express their computational problems in a Grid environment. 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.

Our framework is not bounded to a specific class of applications, does not require new services, and does not necessarily require source code changes. The framework is currently functional on any Grid testbed based on Globus. We believe that is an important advantage because of socio-political issues: cooperation between different organizations, administrators, and users can be very difficult.

5. Experiences.

5.1. The Target Application. We have tested our tool with a Bioinformatics application aimed at predicting the structure and thermodynamic properties of a target protein from its amino acid sequences. The algorithm, tested in the 5th round of Critical Assessment of techniques for protein Structure Prediction (CASP5), aligns with gaps the target sequence with all the 6150 non-redundant structures in the Protein Data Bank (PDB), and evaluates the match between sequence and structure based on a simplified free energy function plus a gap penalty term. The lowest scoring alignment found is regarded as the prediction if it satisfies some quality requirements. For each sequence-structure pair, the search of the optimal alignment is not exhaustive. A large number of alignments are constructed in parallel through a semi-deterministic algorithm, which tries to minimize the scoring function.

To speed up the analysis and reduce the data needed, the PDB files are preprocessed to extract the contact matrices, which provide a reduced representation of protein structures. The algorithm is then applied twice, the first time as a fast search, in order to select the 100 best candidate structures, the second time with parameters allowing a more accurate search of the optimal alignment.

We have applied the algorithm to the prediction of thermodynamic properties of families of orthologous proteins, i.e. proteins performing the same function in different organisms. If a representative structure of this set is known, the algorithm predicts it as the correct structure. The biological results of the comparative study of several proteins are presented elsewhere [19, 7].

5.2. Experiment Preparation. We have modified the application to provide a `restart file` and a `performance profile`. The architecture independent `restart file` stores the best candidate proteins found to that moment and the next protein in the PDB to analyze. The `performance profile` stores the time spent on each iteration of the algorithm, where an iteration consists in the analysis of a given number of sequences.

Table 5.1
The UCM-CAB research testbed.

Name	Architecture	OS	Speed	Memory	Job mgr.	VO
ursa	1×UltraSPARC-IIe	Solaris	500MHz	256MB	fork	UCM
draco	1×UltraSPARC-I	Solaris	167MHz	128MB	fork	UCM
pegasus	1×Pentium 4	Linux	2.4GHz	1GB	fork	UCM
solea	2×UltraSPARC-II	Solaris	296MHz	256MB	fork	UCM
babieca	5×Alpha EV6	Linux	466MHz	256MB	PBS	CAB

Initially, the application does not impose any requirement to the resources, so the `requirement expression` is null. 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 proximity of the candidate resources [16].

The *resource selector* consists of a shell script that queries the MDS [2] for potential execution hosts. 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 performed on each discovered hosts to guarantee user access. Then, the resource is monitored to gather its dynamic status by accessing its local GRIS server. This information is used to assign a rank to each candidate resource based on user-provided preferences. Finally, the resultant prioritized list of candidate resources is used to dispatch the jobs.

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 30 seconds.

The *performance evaluator* is another shell script that parses the `performance profile` and detects performance slowdown when the last iteration time is greater than a given threshold.

The whole experiment was submitted as an array job, where each sequence was analyzed in a separate task of the array, specifying all the needed information in a `job template` file.

The experiment files consists of: the executable (0.5MB) provided for all the resource architectures in the testbed, the PDB files shared and compressed (12.2MB) to reduce the transfer time, the parameter files (1KB), and the file with the sequence to be analyzed (1KB). The final file name of the executable and the file with the sequence to be analyzed is obtained by resolving the variables GW_ARCH and GW_TASK_ID, respectively, at runtime for the current host and job. Input files can be local or remote (specified as a GASS o GridFTP URL), and both can be compressed (to be uncompressed on the selected host) and declared as shared (then stored in the GASS cache and shared by all the jobs submitted to this resource).

5.3. Results on the UCM-CAB Testbed. We have performed the experiments in the UCM-CAB research testbed, which is summarized in table 5.1.

5.3.1. Detection of a Performance Degradation. Let us first consider an experiment consisting in five tasks, each of them applies the structure prediction algorithm to a different sequence of the *ATP Synthase* enzyme (epsilon chain) present in different organisms. Shortly after submitting the experiment, pegasus was overloaded with a compute-intensive application.

Figure 5.1 shows the execution profile in this situation, along with the load in pegasus that caused the performance degradation, and the progress of job 0, obtained from its `performance profile`. Initially four tasks are allocated to babieca and one to pegasus. When the *performance evaluator* detects the performance degradation, it requests a job migration. Since there is a slot available in babieca, the job is migrated to it although it presents lower performance. In spite of the overhead induced by job migration, 6% of the total execution time, job 0 ends before the rest of jobs, because of the better performance offered by pegasus before it became saturated.

5.3.2. Mandatory Change in Resource Requirements. In the following experiment, we have applied the structure prediction algorithm to five sequences of the *Triosephosfate Isomerase* enzyme, which is considerably larger than the previous one, present in different organisms.

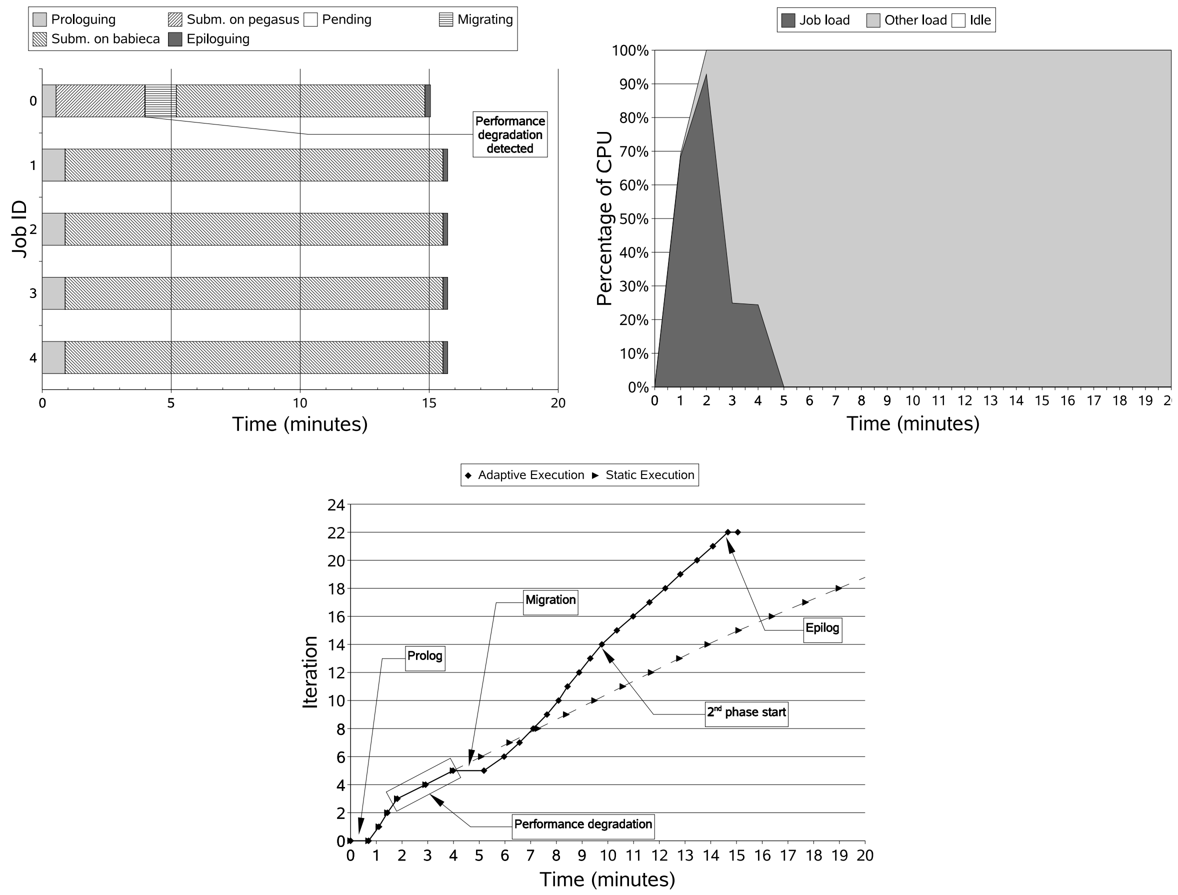

Fig. 5.1. *Execution profile (top), load in pegasus (middle), and progress of job 0 (bottom) when a performance degradation is detected.*

As mentioned in Section 5.1, the target application is divided in two different phases. First, a fair analysis is performed to get the 100 best candidate proteins, and then, a more exhaustive analysis is performed to get the 20 best candidate proteins from the 100 obtained in the first phase. As the second phase analysis performs a more accurate sequence alignment and the target sequence is quite large, it needs more memory than the first phase analysis. Therefore, the application change its resource requirements before starting the second phase to assure that it has enough memory (512MB). The only resource that meets the requirements of the second phase is pegasus.

Figure 5.2 shows the execution profile in this situation. Job 0 starts execution on pegasus, while jobs 1 to 4 start execution on babieca. When job 0 completes its execution, job 1 detects that pegasus has become free and migrates to it, since it presents a better rank (opportunistic job migration). After that, jobs 2 to 4 request a self-migration as they have changed their requirements to complete the second phase of the protein analysis and babieca doesn't meet them. Jobs 0 and 1 also changed their requirements before, but its execution host in that moment (pegasus) met them, so they could continue with their execution. As pegasus is busy with job 1, jobs 2 to 4 have to wait until it becomes available. These jobs are submitted consecutively to pegasus (see figure 5.2) to complete the second phase of the protein analysis.

6. Conclusions. We have shown an effective way for providing adaptive scheduling and execution on Grids. The presented framework does not necessarily require source code changes in the applications, but with minimal changes, applications could benefit from the self-adapting features also provided.

On the scope of the target application, these promising experiments show the potentiality of the Grid to the study of large numbers of protein sequences, and suggests the possible application of this methods to the whole set of proteins in a complete microbial genome.

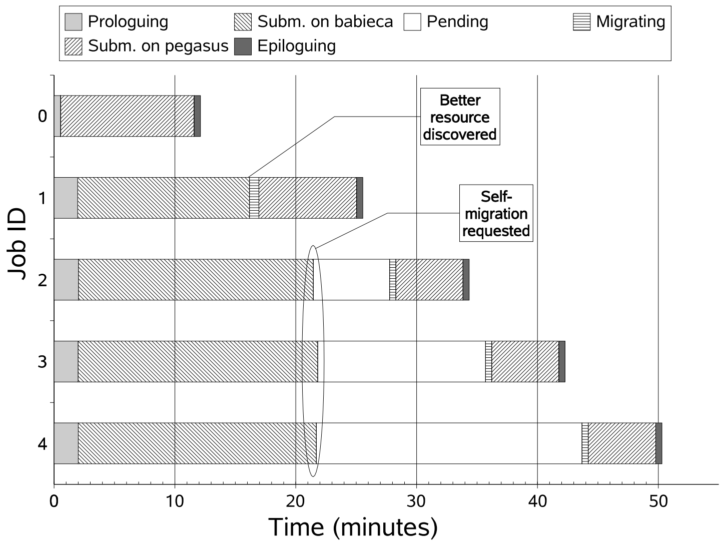

Fig. 5.2. *Execution profile when a mandatory change in resource requirements occurs.*

We are currently working on a *storage resource selector* module to provide support for replica files, specified as a logical file or as a file belonging to a logical collection. In this way the PDB files holding the protein structures, will be scattered on the Grid testbed. The discovery process is performed by accessing the Globus Replica Catalog. The resource selection is based on the proximity between the selected compute resource and the candidate storage resources, along with the values gathered from the MDS GRIS.

Acknowledgments. We would like to thank Ugo Bastolla, staff scientist at the Centro de Astrobiología and developer of the Bioinformatics application used in the experiments, for his support on understanding and modifying the application.

REFERENCES

[1] *Globus Resource Allocation Manager*. http://www.globus.org/gram.

[2] *Monitoring and Discovery Service*. http://www.globus.org/mds.

[3] *The Global Grid Forum*. http://www.gridforum.org.

[4] *The Globus Project*. http://www.globus.org.

[5] G. Allen, E. Seidel, and J. Shalf, *Scientific Computing on the Grid*, Byte, Spring 2002 (2002), pp. 24–32.

[6] M. Baker, R. Buyya, and D. Laforenza, *Grids and Grid Technologies for Wide-Area Distributed Computing*, Intl. J. of Software: Practice and Experience (SPE), 32 (2002), pp. 1437–1466.

[7] U. Bastolla et al., *Reduced Protein Folding Efficiency, Genome Reduction and AT Bias in Obligatory Intracellular Bacteria: An Integrated View*, (2003). (preprint).

[8] F. Berman et al., *The GrADS Project: Software Support for High-Level Grid Application Development*, Intl. J. of High Performance Computing Applications, 15 (2001), pp. 327–34.

[9] ———, *Adaptive Computing on the Grid Using AppLeS*, IEEE Transactions on Parallel and Distributed Systems, 14 (2003), pp. 369–382.

[10] R. Buyya, D.Abramson, and J. Giddy, *A Computational Economy for Grid Computing and its Implementation in the Nimrod-G Resource Broker*, Future Generation Computer Systems, 18 (2002), pp. 1061–1074.

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

[12] 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.

[13] E. Huedo, R. S. Montero, and I. M. Llorente, *A Framework for Adaptive Execution on Grids*, Intl. J. of Software – Practice and Experience, (2004). (in press).

[14] 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.

[15] 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), September 2003.

[16] ———, *Grid Resource Selection for Opportunistic Job Migration*, in Proc. of Intl. Conf. on Parallel and Distributed Computing (Euro-Par 2003), vol. 2790 of Lecture Notes on Computer Science, August 2003, pp. 366–373.

[17] H. Rajic et al., *Distributed Resource Management Application API Specification 1.0*, tech. rep., The Global Grid Forum, 2003. DRMAA Working Group.

[18] J. M. Schopf, *Ten Actions when Superscheduling*, Tech. Rep. GFD-I.4, The Global Grid Forum: Scheduling Working Group, 2001.

[19] R. van Ham et al., *Reductive Genome Evolution in buchnera aphidicola*, Proc. Natl. Acad. Sci. USA, 100 (2003), pp. 581–586.

Edited by: Wilson Rivera, Jaime Seguel.
Received: July 3, 2003.
Accepted: September 1, 2003.