SOFTWARE—PRACTICE AND EXPERIENCE
Softw. Pract. Exper. 2004; **34**:631–651 (DOI: 10.1002/spe.584)

A framework for adaptive execution in grids

Eduardo Huedo[1], Ruben S. Montero[2,*,†] and Ignacio M. Llorente[1,2]

[1]*Centro de Astrobiología, CSIC-INTA, Associated to NASA Astrobiology Institute, 28850 Torrejón de Ardoz, Spain*
[2]*Departamento de Arquitectura de Computadores, Facultad de Informática, Universidad Complutense, Avd. Complutense s/n, 28040 Madrid, Spain*

SUMMARY

Grids offer a dramatic increase in the number of available processing and storing resources that can be delivered to applications. However, efficient job submission and management continue being far from accessible to ordinary scientists and engineers due to their dynamic and complex nature. This paper describes a new Globus based framework that allows an easier and more efficient execution of jobs in a 'submit and forget' fashion. The framework automatically performs the steps involved in job submission and also watches over its efficient execution. In order to obtain a reasonable degree of performance, job execution is adapted to dynamic resource conditions and application demands. Adaptation is achieved by supporting automatic application migration following performance degradation, 'better' resource discovery, requirement change, owner decision or remote resource failure. The framework is currently functional on any Grid testbed based on Globus because it does not require new system software to be installed in the resources. The paper also includes practical experiences of the behavior of our framework on the TRGP and UCM-CAB testbeds. Copyright © 2004 John Wiley & Sons, Ltd.

KEY WORDS: Grid technology; adaptive execution; job migration; Globus

1. INTRODUCTION

Several research centers share their computing resources in Grids, which offer a dramatic increase in the number of available processing and storing resources that can be delivered to applications. Their goal is to provide the end user with a performance higher than that achievable on any single center. These Grids provide a way to access the resources needed for executing the compute and data intensive applications required in several research and engineering fields.

*Correspondence to: Ruben S. Montero, Departamento de Arquitectura de Computadores, Facultad de Informática, Universidad Complutense, Avd. Complutense s/n, 28040 Madrid, Spain.
†E-mail: rubensm@dacya.ucm.es

Contract/grant sponsor: National Aeronautics and Space Administration; contract/grant number: NAS1-97046
Contract/grant sponsor: Ministerio de Ciencia y Tecnología; contract/grant number: TIC 2002-00334
Contract/grant sponsor: Instituto Nacional de Técnica Aeroespacial (INTA)

Received 4 November 2002
Revised 23 April 2003
Accepted 25 September 2003
Published online 24 March 2004

In spite of the great research effort made in Grid computing, application development and execution in the Grid continue requiring a high level of expertise due to its complex nature. In a Grid scenario, a sequential or parallel job is commonly submitted to a given resource by taking the following path [1].

- *Resource discovery and selection*. Based on a set of job requirements, like operating system or platform architecture, a list of appropriate resources is obtained by accessing an information service mechanism. Then a single resource is selected among the candidate resources in the list.
- *Remote system preparation*. The selected host is prepared for job execution. This step usually requires staging of executable and input files.
- *Job submission and migration*. The job is submitted to the selected resource. However, the user may decide to restart its job on a different resource, if a performance slowdown is detected or a 'better' resource is discovered.
- *Job Monitoring*. The job evolution is monitored over time.
- *Termination*. When the job is finished, its owner is notified and some completion tasks, such as output file staging and cleanup, are performed.

The Globus toolkit [2] has become a *de facto* standard in Grid computing. Globus is a core Grid middleware that provides the following components, which can be used separately or together, to support Grid applications: GRAM (Grid Resource Access and Management), GASS (Global Access to Secondary Storage), GSI (Grid Security Infrastructure), MDS (Monitoring and Discovery Service), and Data Grid (GridFTP, Replica Catalog, and Replica Management). These services allow secure and transparent access to resources across multiple administrative domains, and serve as building blocks to implement the stages of Grid scheduling mentioned before. However, the user is responsible for manually performing all the submission steps in order to achieve any functionality [1,3]. Moreover, the Globus toolkit does not provide any native support for job migration and therefore for adaptive execution.

The aim of this paper is to describe a framework that automatically performs all the submission steps and also provides the runtime mechanisms needed for dynamically adapting the application execution. This framework has been developed in the context of the GridWay project, whose aim is to develop user-level tools to reduce the gap between Grid middleware and application developers. We concentrate on the practical issues of its design, implementation and evaluation. Firstly we present a new application model proposed for adaptation to dynamic Grid conditions. Then we describe the architecture of a framework able to support the execution of these Grid-aware applications. The following sections describe how the submission stages are incorporated into the framework. The circumstances under which a migration could be initiated are also discussed. The framework is next compared with other similar approaches. Finally, we demonstrate the capabilities of our framework when managing the execution of a computational fluid dynamics (CFD) code on the TRGP (Tidewater Research Grid Partnership) and UCM-CAB testbeds.

A GRID-AWARE APPLICATION MODEL

Probably, one of the most challenging problems that the Grid computing community has to deal with is the fact that Grids are highly dynamic environments. An application should be able to adapt itself to rapidly changing resource conditions, as follows.

- *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 cancelled 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. Where local jobmanagers do not guarantee exclusive access to compute resources, this may result in initially idle hosts becoming saturated, and *vice versa*. Alternatively, in dedicated batch systems, the saturation of the resource may increase the queue wait time to an unacceptable value.
- *Dynamic resource cost*: In an economy driven grid [4], resource prices can vary depending on the time of the day (working/non-working time) or the resource load (peak/off-peak).

Therefore, in order to obtain a reasonable degree of both application performance and fault tolerance, a job must be able to adapt itself according to the availability of the resources and the current performance provided by them. The emerging Grid technology has led to a new generation of applications that relies on its ability to adapt its execution to dynamic Grid conditions [5]. These new applications must be able to seek out computational resources that fit their needs as their execution evolves. For instance, adaptive-mesh refinement numerical methods systematically refine the computational mesh in those areas where a higher resolution is needed. In this sense, the amount of RAM memory needed to store the computational mesh is not known beforehand. The application must adapt itself to its new requirements, migrating to a resource that provides an adequate amount of RAM memory.

The fundamental aspect of adaptive execution is the recognition of changing conditions of both Grid resources and application demands. Consequently, the classical application model must be modified in order to equip it with such functionality.

- In order to adapt the execution of a job to its dynamic demands, an application must be able to specify its requirements, for example through a requirement expression that contains the attributes that must be met by the target resources. The application could define its initial requirements and dynamically change them when more, or even less, resources (memory, disk, etc.) and different resources (software or license availability) are required.
- Additionally, a procedure to prioritize the resources that fulfill the job requirements must be supplied, for example through a rank expression to dynamically assign a rank to each resource according to the specific needs of each application. The rank expression of a compute-intensive job will assign a higher rank to those hosts with faster CPUs, while a data-intensive application could benefit those hosts near to the input data.
- The performance offered by a given resource may drastically change during the job life. In order to detect performance slowdown, the application is required to keep a log of its performance activity in terms of application intrinsic metrics. For example, a performance profile 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.
- Restart files are highly advisable if dynamic scheduling is performed. User-level checkpointing managed by the programmer must be implemented because system-level checkpointing is not possible among heterogeneous resources. Migration is commonly implemented by restarting the job on the new candidate host. Therefore, the job should generate restart files at regular intervals

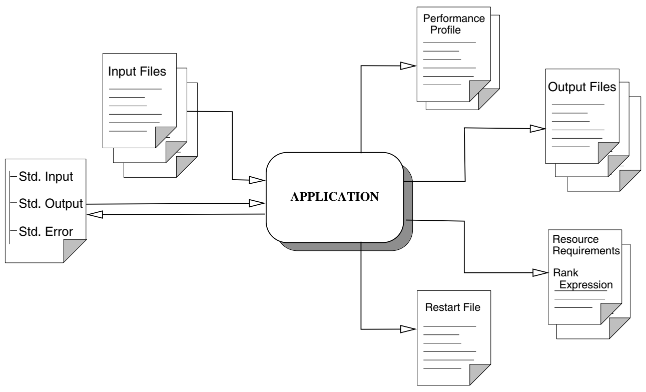


Figure 1. Grid-aware application model.

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. In order not to reduce the number of candidate hosts where a job can migrate, the restart files should be architecture independent.

The Grid-aware application model that we consider is depicted in Figure 1.

ARCHITECTURE OF THE FRAMEWORK

We have developed a framework that provides the support needed to execute Grid-aware applications in dynamic Grids. The core of the framework is a personal submission agent that performs all submission stages and watches over the efficient execution of the job. Adaptation to changing conditions is achieved by dynamic scheduling. Once the job is initially allocated, it is rescheduled when performance slowdown or remote failure are detected, and periodically at each *discovering* interval. Application performance is evaluated periodically at each *monitoring* interval by executing a *performance degradation evaluator* program and by evaluating its accumulated suspension time. A *resource selector* program acts as a personal resource broker to build a prioritized list of candidate resources. Since both programs have access to files dynamically generated by the running job, the application has the ability to take decisions about resource selection and to provide its own performance profile. The submission agent (Figure 2) consists of the following components:

- *request manager*,
- *dispatch manager*,
- *submission manager*,
- *performance monitor*.

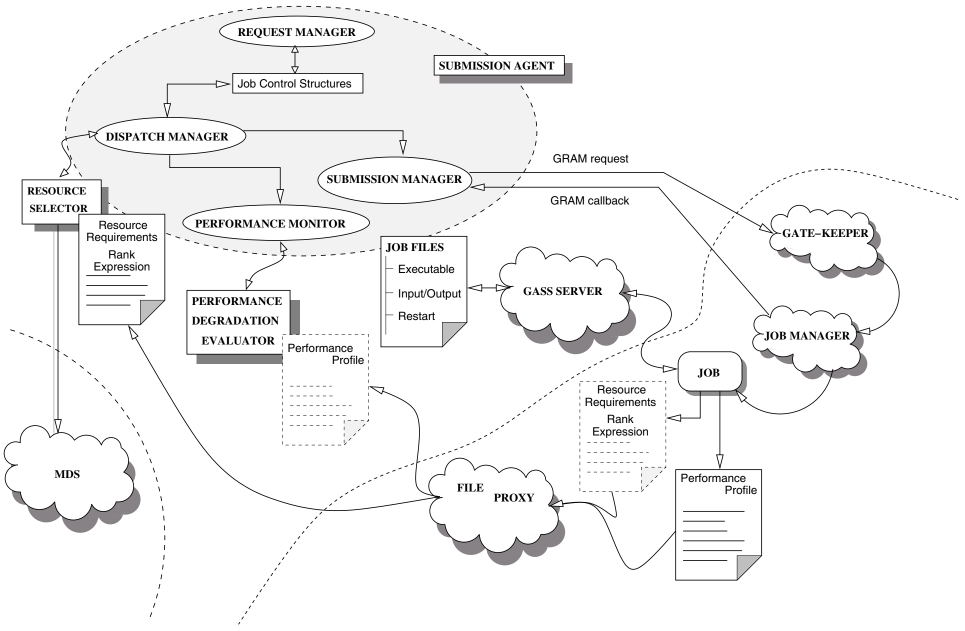


Figure 2. Architecture of the framework.

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*,
- *performance degradation evaluator*,
- *prolog*,
- *wrapper*,
- *epilog*.

The following actions are performed by the submission agent.

- The client application uses a Client API to communicate with the *request manager* in order to submit the job along with its configuration file, or `job template`, which contains all the necessary parameters for its execution. Once submitted, the client may also request control operations to the *request manager*, such as job *stop*/*resume*, *kill* or *reschedule*.
- The *dispatch manager* periodically wakes up at each *scheduling* interval, and tries to submit *pending* and *rescheduled* jobs to Grid resources. It invokes the execution of the *resource selector* corresponding to each job, which returns a prioritized list of candidate hosts. The *dispatch*

manager submits *pending* jobs by invoking a *submission manager*, and also decides if the migration of *rescheduled* jobs is worthwhile or not. If this is the case, the *dispatch manager* triggers a *migration* event along with the new selected resource to the job *submission manager*, which manages the job migration.

- The *submission manager* is responsible for the execution of the job during its lifetime, i.e. until it is *done* or *stopped*. It is initially invoked by the *dispatch manager* along with the first selected host, and is also responsible for performing job migration to a new resource. The Globus management components and protocols are used to support all these actions. The *submission manager* performs the following tasks.
  - *Prologing*: preparing the RSL and submitting the *prolog* executable. The *Prolog* sets up a remote system, transfers executable and input files, and, in the case of restart execution, also transfers restart files.
  - *Submitting*: preparing the RSL, submitting the *wrapper* executable, monitoring its correct execution (as explained in subsequent sections), updating the submission states via Globus callbacks and waiting for *migration*, *stop* or *kill* events from the *dispatch manager*. The *wrapper* wraps the actual job in order to capture its exit code.
  - *Canceling*: canceling the submitted job if a *migration*, *stop* or *kill* event is received by the *submission manager*.
  - *Epiloging*: preparing the RSL and submitting the *epilog* executable. The *epilog* transfers back output files on termination or restart files on migration, and cleans up the remote system.
- The *performance monitor* periodically wakes up at each *monitoring* interval. It requests *rescheduling* actions to detect 'better' resources when performance slowdown is detected and at each *discovering* interval.

RESOURCE DISCOVERY AND SELECTION

Due to the heterogeneous and dynamic nature of the Grid, the end-user must establish the requirements that one to 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 MDS. Usually, resource discovery is only based on static attributes (operating system, architecture, memory size, etc.) collected from the Grid Information Index Service (GIIS), while resource selection is based on dynamic attributes (disk space, processor load, free memory, etc.) that can be obtained from the Grid Resource Information Service (GRIS) or by accessing the Network Weather Service (NWS) [6].

The resource selector executable

The *resource selector* is executed by the *dispatch manager* in order to get a ranked list of candidate hosts when the job is *pending* to be submitted or a *rescheduling* action has been requested. The *resource selector* is a script or a binary executable specified in the `job template`. Its standard output must show a candidate resource per line in a specific fixed format that includes: staging GRAM Job Manager,

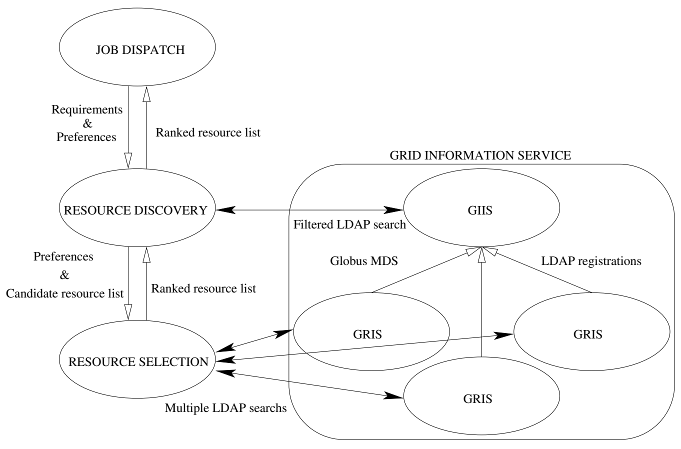


Figure 3. The brokering process scheme of the GridWay framework.

execution GRAM Job Manager, rank, number of slots, and architecture. The job is always submitted to the default queue. However, the *resource selector* should verify that the resource keeps a candidate queue according to the job limits.

This modular approach guarantees the extensibility of the resource selection. Different strategies for application level scheduling can be implemented, from the simplest one based on a pre-defined list of hosts to more advanced strategies based on requirement filters, authorization filters and rank expressions in terms of performance models [7,8]. The `job template` may include input parameters needed for resource discovery and selection, such as `host requirement` and `rank expression` files. This mechanism allows the application to be aware of the Grid environment and take dynamically decisions about its own resource selection, since the `host requirements` and `rank expression` files could be dynamically generated by the running job.

The brokering process of the GridWay framework used in the following experiments is shown in Figure 3. Initially, available compute resources are discovered by accessing the GIIS server and those resources that do not meet the user-provided `host 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 and the available GRAM jobmanagers, are gathered from its local GRIS server. This information is used by a 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.

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 with the cost of accessing the MDS, which can be extremely high [9]. In order to reduce the information retrieval overhead, the GIIS and GRIS information are locally cached at the client host.

Scheduling policy

The goal of the *resource selector* is to find a host that minimizes total response time (file transfer and job execution). Consequently, our application level scheduler promotes the performance of each individual application [10] without considering the rest of *pending*, *rescheduled* or *submitted* applications. This greedy approach is similar to the one provided by most of the local distributed resource management tools [11] and Grid projects [7,8,12]. The remote resources are 'flooded' with requests and subsequent monitoring of performance degradation allows a better balance by migration.

It is well known that this is not the best approach to improve the productivity of the Grid in terms of the number of jobs executed per time unit because it does not balance the interests of different applications [13]. Efficient application performance and efficient system performance are not necessarily the same. For example, when competing applications are executing, the scheduler should give priority to short or new jobs by temporally stopping longer jobs.

Although currently not provided, the *dispatch manager* could make decisions taking into account all *pending*, *rescheduled* and *submitted* jobs with the aim of making an intelligent collective scheduling of them, that is, a user level scheduling approach. Collective scheduling becomes highly important when dealing with parametric jobs. The *resource selector* could also communicate with higher-level schedulers or metaschedulers to take into account system level considerations [13].

REMOTE SYSTEM PREPARATION

Dynamic files are those that are generated or modified on the remote host by the running job and so have to be accessible to the local host. Examples of these files are `host requirement`, `rank expression` or `performance profile` files which could be needed to support dynamic resource selection and performance monitoring. By contrast, the files which do not have to be accessible to the local host (submission client) during job execution on a remote host are referred to here as *static files*. Examples of these files are input, output, and restart files.

Static file management

Data transfer of static files is performed in two steps.

- *Prolog*. The *prolog* module 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 gsiftp URL. Once the files are transferred to the remote host, they are added to the GASS cache so they can be re-used if they are shared with other jobs.
- *Epilog*. The *epilog* module is responsible for transferring back output or restart files, and cleaning up the remote experiment directory. At this point, the files are also removed from the GASS cache.

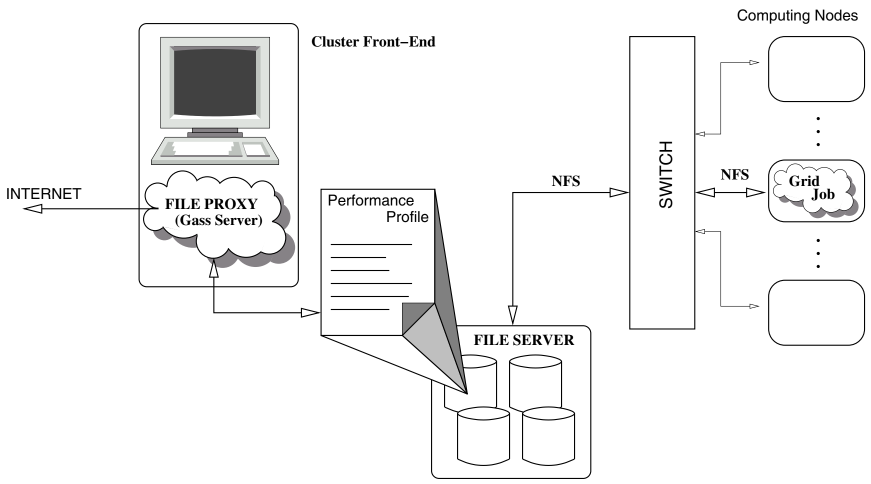


Figure 4. Access to dynamic files on closed systems through a file proxy.

These file transfers are performed through a reverse-server model. The file server (GASS or GridFTP) is started on the local system, and the transfer is initiated on the remote system using Globus transfer tools (i.e. `globus-url-copy` command).

The *prolog* and *epilog* executables[‡] are always submitted to the fork GRAM Job Manager. In this way, our tool is well suited for closed systems such as clusters, where only the front-end node is connected to the Internet and the computing nodes are connected to a system area network, so they are not accessible by the client. Other submission toolkits (Nimrod/G [4] or EDG JSS [14]) only use one executable, or job wrapper, to set up the remote system, transfer files, run the executable and retrieve results. A comparison between both alternatives can be found in [15]. We would like to mention that the new GRAM 1.6 [16] component, introduced in Globus 2.2, provides some support for file staging with job submission.

Dynamic file management

Dynamic file transferring is not always possible through a reverse-server model. Closed systems prevent jobs running in 'private' computational nodes from updating files in the 'public' client host. This problem has been overcome by using a *file proxy* (i.e. GASS or GridFTP server) on the front-end node of the remote system. In this way, the running job updates its dynamic files locally within the cluster, via for example NFS, and they are accessible to the client host through the remote file proxy (Figure 4).

[‡]Some of these scripts are available at http://www.dacya.ucm.es/asds/GridWayPWE.html.

JOB SUBMISSION

The submission agent uses a wrapper to submit the job on a remote host. The *wrapper* executes the submitted job and writes its exit code to standard output, so the submission agent can read it via GASS and can be used to determine whether the job was successfully executed. It is interesting to note that Globus toolkit does not provide any mechanism to capture the exit code of a job. Three situations will be considered, namely

- the exit code is set to a zero value: the job is considered to be *done* with a *success* status;
- the exit code is set to a non-zero value: the job is considered to be *done* with a *failed* status;
- the exit code is not set: the job has been *canceled* and, consequently, a job *rescheduling* action is requested.

The capture of the remote execution exit code allows users to define complex jobs, where each depends on the output and exit code from the previous job. They may even involve branching, spawning and loops, allowing the exploitation of the parallelism on the work flow of certain types of applications.

JOB MIGRATION

A Grid environment presents unpredictable changing conditions, such as dynamic resource load, high fault rate, or continuous addition and removal of resources. Migration is the key issue for adaptive execution of jobs on dynamic Grid environments. In our work, we have considered the following circumstances, related to the situations discussed previously, under which a migration could be initiated.

1. *Grid initiated migration*:
    - a new 'better' resource is discovered (opportunistic migration);
    - the remote resource or its network connection fails;
    - the submitted job is canceled or suspended by the resource administrator.
2. *Application initiated migration*:
    - the application detects performance degradation or performance contract violation;
    - self-migration when the resource requirements of the application change.
3. *User initiated migration*:
    - the user explicitly requests a job migration.

Rescheduling policies

Our framework currently considers the following reasons for rescheduling, i.e. situations under which a *migration* event could be triggered.

1. The *request manager* receives a *rescheduling* request from the user.
2. The *performance monitor* requests a *rescheduling* action at each *discovering* interval to detect a 'better' resource.

3. The *performance monitor* requests a re-scheduling action in a given *monitoring* interval:
    (a) if the *performance degradation evaluator* detects a performance slowdown;
    (b) if the *suspension time* exceeds a given *maximum suspension time* threshold; the *submission manager* takes count of the accumulated suspension time spent by the job on *pending* and *suspended* Globus states.
4. The *submission manager* detects a failure:
    (a) cancellation, detected through the job exit code obtained from the *wrapper* standard output; or premature termination, detected when a Globus failed callback is received;
    (b) remote host or network crashes, its detection is explained in the next section.

The reason for rescheduling is evaluated 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* with a new selected host, even if the new host presents lower rank than the current one. 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. Other conditions [17], apart from the reason for rescheduling and the rank of the new selected host, could also be evaluated: time to finalize [17,18], input and restart file transfer costs [18] etc.

When a migration order is finally granted, the *submission manager* cancels the job (if it is still running), invokes the *epilog* on the current host (if the files are accessible) and the *prolog* on the new remote host. The local host always keeps the last checkpoint files in case the connection with the remote host fails. Due to the size of the checkpoint files, migration may be an expensive operation that is not suitable for a given class of applications or situations.

JOB MONITORING

Our framework provides two mechanisms to detect performance slowdown.

- A *performance degradation evaluator* is periodically executed at each *monitoring* interval by the *performance monitor* to evaluate a rescheduling condition. Different strategies could be implemented, from the simplest one based on querying the Grid information system about workload parameters to more advanced strategies based on detection of performance contract violations [19]. The *performance degradation evaluator* is a script or a binary executable specified in the `job template`, which can also include additional parameters needed for the performance evaluation. A mechanism to deal with application own metrics is provided since the files processed by the *Performance Degradation Evaluator* could be dynamically generated by the running job. The rescheduling condition verified by the *performance degradation evaluator* could be based on the performance history using advanced methods like fuzzy logic, or comparing the performance with the initial performance attained, or a base performance.
- A running job could be temporally suspended by the resource administrator or by the local queue scheduler on the remote resource. The submission agent takes account of the overall *suspension time* of its job and requests a *rescheduling* action if it exceeds a give threshold. Notice that the *maximum suspension time* threshold is only effective on queue-based resource managers.

In order to detect remote failure, we have followed an approach similar to that provided by Condor/G [20]. The GRAM Job Manager is probed by the submission agent periodically at each *polling* interval. If the GRAM Job Manager does not respond, the GRAM Gatekeeper is probed. If the GRAM Gatekeeper responds, a new GRAM Job Manager is started to resume watching over the job. If the GRAM Gatekeeper fails to respond, a *rescheduling* action is immediately requested.

RELATED WORK

The management of jobs within the same department is addressed by many research and commercial systems [11]: Condor, Load Sharing Facility, Sun Grid Engine, Portable Batch System, LoadLeveler etc. Some of these tools, such as Sun Grid Engine Enterprise Edition [21], also allow the interconnection of multiple departments within the same administrative domain, which is called enterprise interconnection, as long as they run the same distributed resource management software. Other tools, such as Condor Flocking [22], even allow the interconnection of multiple domains, which is called worldwide interconnection. However, they are unsuitable in computational Grids where resources are scattered across several administrative domains, each with its own security policies and distributed resource management systems.

The Globus middleware [2] provides the services needed to enable secure multiple domain operation with different resource management systems and access policies. There are projects underway, like Condor/G [20], EveryWare [23], AppLeS [24], Nimrod/G [25], or ILab [26], which are developing user-oriented submission tools over the Globus middleware to simplify the efficient exploitation of a computational Grid. All the aforementioned application schedulers share many features, with differences in the way they are implemented [27,28]. We believe that there are no better tools, each focuses on a specific issue and contributing significant improvements in the field.

Adaptive schedulers have been widely investigated in the literature, for example the AppLeS project, when targeting *long-running* applications, supporting schedule adaptation to tolerate changes in resource availabilities and in the dynamic performance exhibited by many Grid resources [29,30]. The Nimrod/G resource broker incorporates schedule adaptation in order to meet several user QoS (quality of service) requirements, like deadline or budget, and also to adapt to resource availability and performance.

Adaptive execution is also being explored in the context of the Grid Application Development Software (GrADS) project [31]. The aim of the GrADS projects is to simplify distributed heterogeneous computing in the same way that the World Wide Web simplified information sharing over the Internet. GrADS provides new external services to be accessed by Grid users and, mainly, by application developers to develop Grid-aware applications. Its execution framework [32] is based on three components: the Configurable Object Program, which contains application code and strategies for application mapping; the Resource Selection Model, which provides estimation of the application performance on specific resources; and the Contract Monitor, which performs job interrupting and remapping when performance degradation is detected. GrADS is an ambitious project that involves several outstanding research groups in Grid technology.

The need for a nomadic migration approach for job execution on a Grid environment has been previously discussed in [33]. The prototype of a migration framework, called the ‘Worm’, was executed on the Egrid testbed [34]. The ‘Worm’ was implemented within the Cactus programming environment. Cactus is an open source problem-solving environment for the construction of parallel solvers for partial differential equations that enables collaborative code development between different groups [35]. The extension of the 'Worm' migration framework to make use of Grid services is described in [33]. Cactus incorporates, through Grid-aware infrastructure thorns [36], adaptive resource selection mechanisms (Resource Locator Service) that allow automatic application migration to 'better' resources (Migrator Service) to deal with changing resource characteristics and adaptive resource selection mechanisms [37]. The adaptation to dynamic Grid environments has been studied by job migration to 'faster/cheaper' systems, considered when better systems are discovered, when requirements change or when job characteristics change.

In the context of the GrADS project, the usefulness of a Grid to solve large numerical problems has been demonstrated by integrating numerical libraries like ScaLAPACK into the GrADS system [38]. The Resource Selector component accesses MDS and NWS to provide the information needed by the Performance Modeler to apply an application-specific execution model and so obtain a list of final candidate hosts. The list is passed to the Contract Developer which approves the contract for the Application Launcher. The submitted application is monitored by the Contract Monitor through the Autopilot manager [39] that can detect contract violations by contacting the sensors and determining if the application behaves as predicted by the model. A migration framework that takes into account both the system load and application characteristics is described in [13].

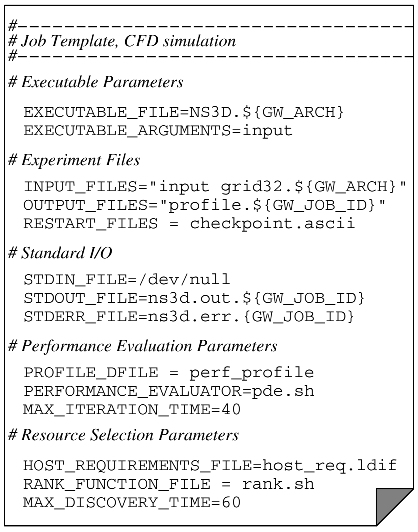
The aim of the GridWay project is similar to that of the GrADS project: to simplify distributed heterogeneous computing. However, its scope is different. Our framework provides a submission agent that incorporates the runtime mechanisms needed for transparently executing jobs in a Grid. In fact, our framework could be used as a building block for much more complex service-oriented Grid scenarios like GrADS. Other projects have also addressed resource selection, data management, and execution adaptation. We do not claim innovation in these areas, but note the advantages of our modular architecture for job adaptation to a dynamic environment.

- It is not bounded to a specific class of application generated by a given programming environment, which extends its application range.
- It does not require new services, which considerably simplify its deployment.
- It does not necessarily require code changes, which allows reusing of existing software.
- It is extensible, which allows its communication with the Grid services available in a given testbed.

We would like to mention that the experimental framework does not require new system software to be installed in the Grid resources. The framework is currently functional on any Grid testbed based on Globus. We believe that this is an important advantage because of socio-political issues; cooperation between different research centers, administrators and users is always difficult.

EXPERIENCES

We next demonstrate the capabilities of the previously described experimental framework. First, we analyze its job adaptation functionality in the execution of a single job. Then, we study the scheduling of a parameter sweep application, where the simulation comprises a moderate to high number of jobs that are to be executed on the Grid.



```
#---------------------------------
# Job Template, CFD simulation
#---------------------------------
# Executable Parameters
  EXECUTABLE_FILE=NS3D.${GW_ARCH}
  EXECUTABLE_ARGUMENTS=input
# Experiment Files
  INPUT_FILES="input grid32.${GW_ARCH}"
  OUTPUT_FILES="profile.${GW_JOB_ID}"
  RESTART_FILES = checkpoint.ascii
# Standard I/O
  STDIN_FILE=/dev/null
  STDOUT_FILE=ns3d.out.${GW_JOB_ID}
  STDERR_FILE=ns3d.err.{GW_JOB_ID}
# Performance Evaluation Parameters
  PROFILE_DFILE = perf_profile
  PERFORMANCE_EVALUATOR=pde.sh
  MAX_ITERATION_TIME=40
# Resource Selection Parameters
  HOST_REQUIREMENTS_FILE=host_req.ldif
  RANK_FUNCTION_FILE = rank.sh
  MAX_DISCOVERY_TIME=60
```

Figure 5. Job template for the target application.

In both situations, the target application solves the 3D incompressible Navier–Stokes equations in the simulation of a boundary layer over a flat plate. The numerical core consists of an iterative robust multigrid algorithm characterized by a compute-intensive execution profile [40,41]. The application generates checkpoint files at each multigrid iteration.

Description of the experiments

A fragment of the `job template` used in the following experiments is shown in Figure 5. The experiment files consist of the executable (2 MB), and the computational mesh (0.5 MB), provided for all the resource architectures in the testbed. The final file names are obtained by resolving the variable `${GW_ARCH}` at runtime for the selected host. The checkpoint file (5 MB) needed to restart job execution in case of job migration is also specified; this file is architecture independent. Once the job finishes, the standard output (8 KB) and the velocity profile (5 KB) at the middle of the flat plate are transferred back to the client. Similarly, the final file names are obtained by resolving the `${GW_JOB_ID}` variable with the job id assigned by the framework.

The resource selection consists of a shell script that queries MDS for potential execution hosts, attending the following criteria.

- Host requirements are specified in a `host requirement` file (`host_req.ldif`), which can be dynamically generated by the running job. The host requirement setting is a LDAP filter, which is used by the *resource selector* to query MDS and so obtain a preliminary list of potential hosts. In the experiments below, we will impose a minimum main memory of 512 MB, enough to accommodate the CFD simulation. The `host requirement` file will be of the form:

```
(Mds-Memory-Ram-freeMB>=512)
```

- A rank (`rank.sh`) is assigned to each potential host based on MDS attributes. Since our target application is a computing-intensive simulation, the rank expression benefits those hosts with less workload and so better performance. The following expression was considered:

$$rank = \begin{cases} FLOPS & \text{if } CPU_{15} \geq 1 \\ FLOPS \cdot CPU_{15} & \text{if } CPU_{15} < 1 \end{cases} \quad (1)$$

  where *FLOPS* is the peak performance achievable by the host CPU, and CPU_{15} is the total free CPU time in the last 15 min. It is interesting to note that in the case of heterogeneous clusters *FLOPS* is the average performance of all computing nodes. However, other alternatives have been proposed in the literature. For example, in the NorduGrid project [42] a more conservative approach is taken, equaling *FLOPS* in the performance of the slowest node.

The performance of the target application is based on the time spent in each multigrid iteration. The time consumed in each iteration is appended by the running job to a `performance profile` file (`perf_profile`) specified as dynamic in the `job template`. The *performance degradation evaluator* verifies at each *monitoring* interval if the time consumed in each iteration is higher than a given threshold (`MAX_ITERATION_TIME`). This performance contract (iteration threshold) and contract monitor (*performance degradation evaluator*) are similar to those used in [37].

Job adaptation to dynamic Grid conditions

We now consider some situations where job execution can take advantage of the migration capabilities of the GridWay framework. The scenarios described below were artificially created. However, they try to resemble real situations that may happen in a Grid. The experiments were conducted on the Tidewater Research Grid Partnership (TRGP) [43] testbed.

The TRGP was started in summer 2001 to foster the development and use of Grid computing for a variety of applications in the computational sciences. TRGP members at the end of August 2002, when the experimental results were taken, included ICASE and the College of William and Mary. The TRGP Grid is highly heterogeneous, see Table I for a summary of the characteristics of each resource. The interconnection between both sites is performed by a 'public' non-dedicated network. The ICASE workstations and servers are interconnected by a Fast Ethernet switched network.

Periodic rescheduling to detect new resources

In this case, the *discovering* interval has been deliberately set to a small value (60 s) in order to quickly re-evaluate the performance of the resources. The execution profile of the application is presented in Figure 6. Initially, only ICASE hosts are available for job submission, since sciclone has been shutdown for maintenance. The *resource selector* chooses urchin to execute the job, and the files are

Table I. Summary of the TRGP resource characteristics.

Resource name	VO	CPU architecture	Peak performance	Nodes	OS	Memory	GRAM
sciclone	W&M	Sun UltraSPARC	115 Gflops	160	Solaris 8	54 GB	PBS
coral	ICASE	Intel Pentium	89 Gflops	68	Linux 2.4	56 GB	PBS
whale	ICASE	Sun UltraSPARC	1.8 Gflops	2	Solaris 7	4 GB	fork
urchin	ICASE	Sun UltraSPARC	672 Mflops	2	Solaris 7	1 GB	fork
carp	ICASE	Sun UltraSPARC	900 Mflops	1	Solaris 7	256 MB	fork
tetra	ICASE	Sun UltraSPARC	800 Mflops	1	Solaris 7	256 MB	fork
bonito	ICASE	Sun UltraSPARC	720 Mflops	1	Solaris 7	256 MB	fork

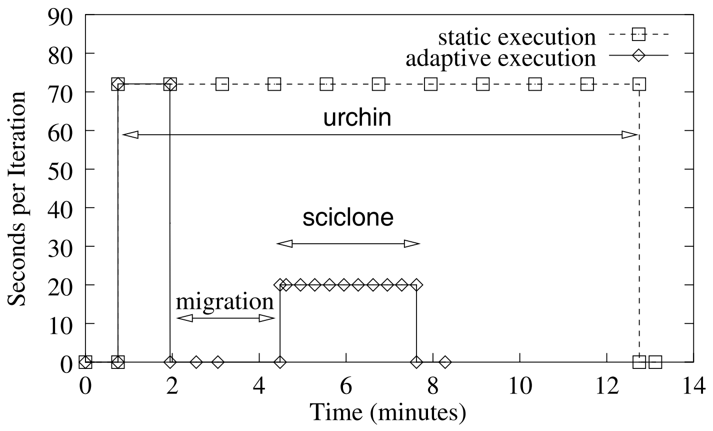


Figure 6. Execution profile of the CFD code when a new 'better' resource is detected.

transferred (prolog and submission in time steps 0–34 s). The job starts executing at time step 34 s. A *discovering* period expires at time step 120 s and the *resource selector* finds sciclone to present higher rank than the original host (time steps 120–142 s). The migration process is then initiated (cancellation, epilog, prolog and submission in time steps 142–236 s). Finally the job completes its execution on sciclone. Figure 6 shows how the overall execution time is 42% lower when the job is migrated. Note that migration time, 95 s, is about 20% of the overall execution time.

Performance degradation detected using a `performance profile` *dynamic file*

The *resource selector* finds whale to be the best resource, and the job is submitted (prolog and submission in time steps 0–34 s). However, whale is overloaded with a compute-intensive workload at time step 34 s. As a result, a performance degradation is detected when the iteration time exceeds the

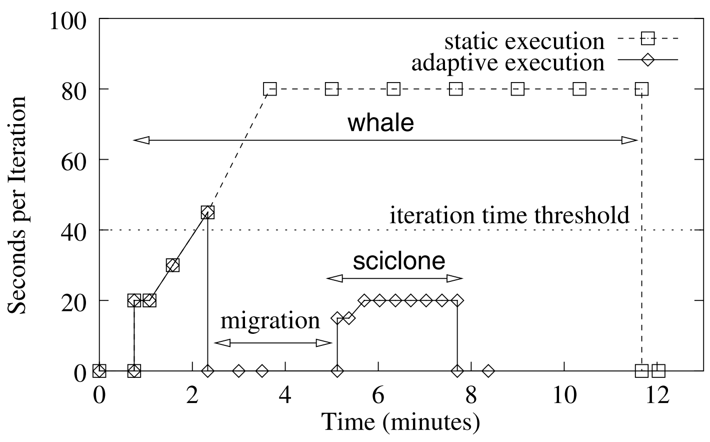


Figure 7. Execution profile of the application when an artificial workload is executed.

iteration time threshold (40 s) at time step 209 s. The job is then migrated to sciclone (cancellation, epilog, prolog and submission in time steps 209–304 s), where it continues executing from the last checkpoint context. The execution profile for this situation is presented in Figure 7. In this case the overall execution time is 35% lower when the job is migrated. The cost of migration, 95 s, is about 21% of the execution time. The speed-up obtained through job migration strongly depends on the amount of the computational work already performed by the application, and the overhead induced by job migration [17,18].

Adaptive scheduling of a parameter sweep application

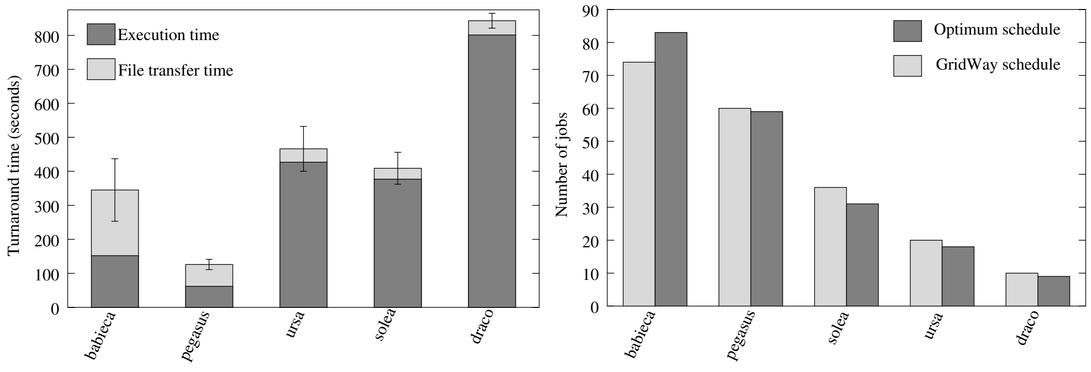
Let us now consider a parameter sweep application consisting of 200 independent tasks. Each task calculates the flow over a flat plate for a different Reynolds number, ranging from 10^2 to 10^4. In this experiment we have used the UCM-CAB testbed, whose main characteristics are summarized in Table II.

The overall execution time for parameter sweep application was 8778 s, with an average job turnaround time of 43 s. Figure 8(a) presents the average job turnaround time on each host of the UCM-CAB grid; error bars represent the standard deviation of the turnaround time. These times include the overhead induced by the Globus middleware. In addition, the average execution time on babieca includes the queue wait time on the PBS batch system. Compared with the single host execution on the fastest machine in the testbed (pegasus, 62 s per job), these results represent a 30% reduction in the overall execution time.

We will next evaluate the schedule performed by the GridWay framework compared with the optimum *static* schedule. This comparison can only be seen as a reference, the main goals of which are to establish an upper performance bound, and, to highlight the relevance of adaptive scheduling in

Table II. Summary of the UCM-CAB grid resource characteristics.

Resource name	VO	Model	Speed	Nodes	OS	Memory	GRAM
ursa	UCM	Sun Blade 100	500 MHz	1	Solaris 8	256 MB	fork
draco	UCM	Sun Ultra 1	167 MHz	1	Solaris 8	128 MB	fork
pegasus	UCM	Intel Pentium 4	2.4 GHz	1	Linux 2.4	1 GB	fork
solea	UCM	Sun Enterprise 250	296 MHz	2	Solaris 8	256 MB	fork
babieca	CAB	Compaq Alpha DS10	466 MHz	4	Linux 2.4	1 GB (total)	PBS



(a) (b)

Figure 8. Average and standard deviation in turnaround time for the parameter sweep application on each host of the testbed (a). Number of jobs scheduled on each host by the optimum and GridWay schedules (b).

a Grid environment. The optimum grid schedule will minimize the makespan of the application [44]:

$$\max\{N_i\overline{T}_i\}i = \{\text{pegasus, draco, solea, ursa, babieca}\} \qquad (2)$$

where N_i is the number of jobs executed on host i, and $\overline{T}_i$ is the average job turnaround time on host i. Figure 8(b) shows the number of jobs scheduled on each host by the GridWay framework, and the solution to the optimization problem 2. The overall execution time for the optimum schedule is 7285 s, with an average job turnaround time of 36 s, 17% better than the schedule made by the GridWay framework.

The main difference between both schedules is the greater number of jobs allocated to babieca by the optimum schedule. Fourteen percent of the jobs submitted to babieca failed, and so they had to be dynamically rescheduled to other available hosts in the testbed. Figure 9 shows the dynamic job turnaround time during the execution of the parameter sweep application. As could be expected the failure experimented on babieca increases the average and dynamic job turnaround times, as well as the overall execution time.

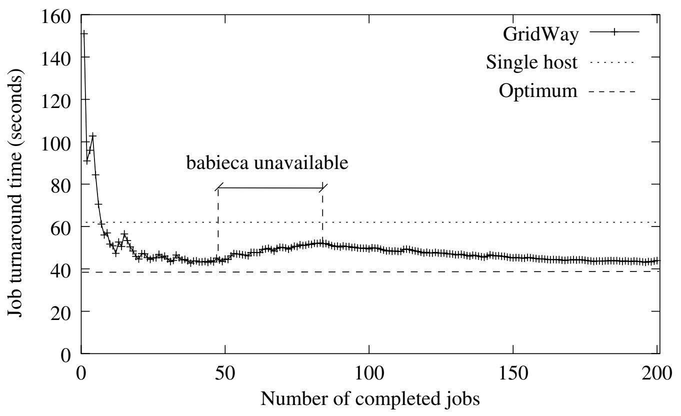


Figure 9. Dynamic job turnaround time in the execution of the parameter sweep application.

CONCLUSIONS

The exploitation in an aggregated form of geographically distributed resources is far from being straightforward. In fact, the Grid will be for many years a challenging area due to its dynamic and complex nature. We believe that the submission framework presented in this paper is a step forward in insulating the application developer from the complexity of the Grid submission. The framework provides the runtime mechanisms needed for dynamically adapting an application to a changing Grid environments. The core of the framework is a personal submission agent that transparently performs all submission stages and watches over the efficient execution of the job. Adaptation to the dynamic nature of the Grid is achieved by implementing automatic application migration following performance degradation, 'better' resource discovery, requirement change, owner decision or remote resource failure. The application has the ability to take decisions about resource selection and to self-migrate to a new resource.

The experimental results are promising because they show how application adaptation achieves enhanced performance. Both the fault tolerance and the response time are improved when the application is submitted through the framework. Simultaneous submission of several applications in order to harness the highly distributed computing resources provided by a Grid is also demonstrated for a parameter sweep application. Our framework is able to efficiently manage applications suitable to be executed on dynamic conditions.

The framework design and implementation are based on a modular architecture to allow extensibility of its capabilities. An experimental user can incorporate different resource selection and performance degradation strategies to perform job adaptation. In fact, the framework could be used as a testbed to evaluate different dynamic scheduling, file transferring, monitoring and adaptation strategies. The Grid-aware application description and the way in which the submission stages are performed could also serve as practical models for future work in the field.

ACKNOWLEDGEMENTS

This work has been performed in part using computational facilities at the College of William and Mary, and was enabled by grants from Sun Microsystems, the National Science Foundation, and Virginia's Commonwealth Technology Research Fund. We would like to thank Thomas Eidson and Tom Crockett for their help with TRGP.

REFERENCES

1. Schopf JM. Ten actions when Grid scheduling: The user as Grid scheduler. *Grid Resource Management: State of the Art and Future Trends*, ch. 2, Nabrzyski J, Schopf JM, Weglarz J (eds.). Kluwer Academic: Boston, MA, 2003.
2. Foster I, Kesselman C. Globus: A metacomputing infrastructure toolkit. *International Journal of Supercomputer Applications* 1997; **11**(2):115–128.
3. Schopf JM, Nitzberg B. Grids: The top ten questions. *Scientific Programming (Grid Computing)* **10**(2):103–111.
4. Buyya R, Abramson D, Giddy J. A computational economy for Grid computing and its implementation in the Nimrod-G resource broker. *Future Generation Computer Systems* 2002; **18**(8):1061–1074.
5. Allen G, Seidel E, Shalf J. Scientific computing on the Grid. *Byte* 2002; **24**(Spring):24–32.
6. Wolski R, Spring N, Hayes J. The Network Weather Service: A distributed resource performance forecasting service for metacomputing. *Journal of Future Generation Computing Systems* 1999; **15**(5–6):757–768.
7. Liu C, Yang L, Foster I, Angulo D. Design and evaluation of a resource selection framework for Grid applications. *Proceedings 11th IEEE Symposium on High-Performance Distributed Computing*. IEEE Computer Society: Los Alamitos, CA, 2002; 63–72.
8. Dail H, Berman F, Casanova H. A decoupled scheduling approach for Grid application development environments. *Journal of Parallel and Distributed Computing* 2003; **63**(5):505–524.
9. Dail H, Casanova H, Berman F. A decoupled scheduling approach for the GrADS environment. *Proceedings SuperComputing (SC02)*, Baltimore, MD, 2002. IEEE Computer Society: Los Alamitos, CA, 2002; 1–14.
10. Foster I, Kesselman C. *The Grid: Blueprint for a New Computing Infrastructure*. Morgan-Kaufman, 1999.
11. El-Ghazawi T, Gaj K, Alexandinis N, Schott B. Conceptual comparative study of job management systems. *Technical Report*, George Mason University, February 2001.
12. WP1 Workload Management Software. http://server11.infn.it/workload-grid/documents.html [2002].
13. Vadhiyar SS, Dongarra JJ. A metascheduler for the Grid. *Proceedings 11th IEEE International Symposium on High Performance Distributed Computing (HPDC'02)*, Edinburgh, July 2002. IEEE Computer Society: Los Alamitos, CA, 2002; 343–351.
14. European DataGrid Workload Management Work Package. http://server11.infn.it/workload-grid/ [2002].
15. Comparison of HTB and Nimrod/G: Job Dispatch. http://www-unix.mcs.anl.gov/~giddy/comp.html [2002].
16. Globus Resource Allocation Manager (gram 1.6) Documentation. http://www-unix.globus.org/api/c-globus-2.2/globus_gram_documentation/html [2002].
17. Vadhiyar S, Dongarra J. A performance oriented migration framework for the Grid. *Proceedings 3rd IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid)*. IEEE Computer Society: Los Alamitos, CA, 2003; 130–137.
18. Huedo E, Montero RS, Llorente IM. Experiences on Grid resource selection considering resource proximity. *Proceedings First European Across Grids Conference* (*Lecture Notes in Computer Science*). Springer. To appear.
19. Vraalsen F, Aydt RA, Mendes CL, Reed DA. Performance contracts: Predicting and monitoring Grid application behavior. *Proceedings of the 2nd International Workshop on Grid Computing (GRID2001)*, Denver, CO, November 2001 (*Lecture Notes in Computer Science*, vol. 2242). Springer: Heidelberg, 2001; 154–165.
20. Frey J, Tannenbaum T, Livny M, Foster I, Tuecke S. Condor-G: A computation management agent for multi-institutional Grids. *Cluster Computing* 2002; **5**(3):237–246.
21. How Sun Grid Engine, Enterprise Edition 5.3 works. *Technical Report, Sun Microsystems White Paper*, Sun Microsystems, Santa Clara, CA, 2002. http://wwws.sun.com/software/gridware/sgeee53/wp-sgeee/index.html.
22. Evers X, de Jongh JFCM, Boontje R, Epema DHJ, van Dantzig R. Condor flocking: Load sharing between pools of workstations. *Technical Report DUT-TWI-93-104*, Delft University of Technology, The Netherlands, 1993.
23. Wolski R, Brevik J, Krintz C, Obertelli G, Spring N, Su A. Running everyware on the computational Grid. *Proceedings of SuperComputing (SC99)*, Portland, OR, November, 1999. ACM Press: New York, 1999.
24. Casanova H, Legrand A, Zagorodnov D, Berman F. Heuristics for scheduling parameter sweep applications in Grid environments. *Proceedings 9th Heterogeneous Computing Workshop (HCW2000)*, Cancun, Mexico, May 2000. IEEE Computer Society: Los Alamitos, CA, 2000; 349–363.

25. Buyya R, Abramson D, Giddy J. Nimrod/G: An architecture for a resource management and scheduling system in a global computation Grid. *Proceedings 4th IEEE International Conference on High Performance Computing in Asia-Pacific Region (HPC Asia)*, Beijing, China, 2000. IEEE Computer Society: Los Alamitos, CA, 2000.
26. Yarrow M, McCann KM, Biswas R, Van der Wijngaart RF. ILab: An advanced user interface approach for complex parameter study process specification on the information power Grid. *Proceedings Grid 2000: First IEEE/ACM International Workshop*, Bangalore, India, 2000 (*Lecture Notes in Computer Science*, vol. 1971). Springer: Heidelberg, 2000; 146–157.
27. Baker M, Buyya R, Laforenza D. Grids and Grid technologies for wide-area distributed computing. *Software—Practice and Experience* 2002; **32**(15):1437–1466.
28. Krauter K, Buyya R, Maheswaran M. A taxonomy and survey of Grid resource management systems for distributed computing. *Software—Practice and Experience* 2002; **32**(2):135–164.
29. Casanova H, Obertelli G, Berman F, Wolski R. The AppLeS parameter sweep template: User-level middleware for the Grid. *Proceedings of Supercomputing 2000 (SC2000)*, Dallas, TX, November 2000. ACM Press: New York, 2000.
30. Berman F, Wolski R, Casanova H, Cirne W, Dail H, Faerman M, Figueira S, Hayes J, Obertelli G, Schopf J, Shao G, Smallen S, Spring N, Su A, Zagorodnov D. Adaptive computing on the Grid using AppLeS. *IEEE Transactions on Parallel and Distributed Systems* 2003; **14**:1–14.
31. Berman F, Chien A, Cooper K, Dongarra J, Foster I, Gannon D, Johnsson L, Kennedy K, Kesselman C, Mellor-Crummey J, Reed D, Torczon L, Wolski R. The GrADS Project: Software support for high-level Grid application development. *International Journal of High Performance Computing Applications* 2001; **15**(4):327–334.
32. Kennedy K *et al.* Toward a framework for preparing and execution adaptive Grid applications. *Proceedings NSF Next Generation Systems Program Workshop, International Parallel and Distributed Processing Symposium*, Fort Lauderdale, FL, April 2002. IEEE Computer Society: Los Alamitos, CA, 2002.
33. Lanfermann G, Allen G, Radke T, Seidel E. Nomadic migration: A new tool for dynamic Grid computing. *Proceedings 10th Symposium on High Performance Distributed Computing (HPDC10)*, San Francisco, CA, August 2001. IEEE Computer Society: Los Alamitos, CA, 2001.
34. Allen G, Benger W, Dramlitsch T, Goodale T, Hege H-C, Lanfermann G, Merzky A, Radke T, Seidel E. Early experiences with the Egrid testbed. *Proceedings IEEE International Symposium on Cluster Computing and the Grid (CCGrid 2001)*, Brisbane, Australia, May 2001. IEEE Computer Society: Los Alamitos, CA, 2001; 130–137.
35. Allen G, Goodale T, Lanfermann G, Radke T, Seidel E. The Cactus code: A problem solving environment for the Grid. *9th IEEE International Symposium on High Performance Distributed Computing (HPDC9)*, Pittsburg, PA, May 2001. IEEE Computer Society: Los Alamitos, CA, 2001.
36. Allen G, Dramlitsch T, Foster I, Goodale T, Karonis N, Ripeanu M, Seidel E, Toonen B. Supporting efficient execution in heterogeneous distributed computing environments with Cactus and Globus. *Proceedings of Supercomputing 2001 (SC2001)*, Denver, CO, November, 2001. ACM Press: New York, 2001.
37. Allen G, Angulo D, Foster I, Lanfermann G, Liu C, Radke T, Seidel E, Shalf J. The Cactus Worm: Experiments with dynamic resource discovery and allocation in a Grid environment. *Proceedings of the European Conference on Parallel Computing (EuroPar) 2001*, Manchester, August 2001 (*Lecture Notes in Computer Science*). Springer: Heidelberg, 2001.
38. Petitet A, Balckford S, Dongarra J, Ellis B, Fagg G, Roche K, Vadhiyar S. Numerical libraries and the Grid: The GrADS experiments with ScaLAPACK. *International Journal of High Performance Computing Applications* 2001; **15**(4):359–374.
39. Ribler RL, Vetter JS, Simitci H, Reed DA. Autopilot: Adaptive control of distributed applications. *Proceedings 7th IEEE International Symposium on High Performance Distributed Computing (HPDC11)*, July 1998. IEEE Computer Society: Los Alamitos, CA, 1998.
40. Montero RS, Llorente IM, Salas MD. Robust multigrid algorithms for the Navier–Stokes equations. *Journal of Computational Physics* 2001; **173**:412–432.
41. Prieto M, Montero RS, Espadas D, Llorente IM, Tirado F. Parallel multigrid for anisotropic elliptic equations. *Journal on Parallel and Distributed Computing* 2001; **199**:543–554.
42. The Nordugrid Project. http://www.nordugrid.org/ [December 2003].
43. Tidewater Research Grid Partnership. http://www.tidewaterrgp.org [October 2002].
44. Casanova H, Legrand A, Zagoridnov D, Berman F. Heuristics for scheduling parameter sweep applications in Grid environments. *Proceedings 9th Heterogeneous Computing Workshop (HCW01)*, Cancun, Mexico, 2000. IEEE Computer Society: Los Alamitos, CA, 2000; 349–363.