An Object-Oriented View of Grid Computing Technologies to Abstract Remote Task Execution

J. M. Alonso, V. Hernández, G. Moltó
Departamento de Sistemas Informáticos y Computación
Universidad Politécnica de Valencia
Camino de Vera s/n, 46022 Valencia, Spain
{jmalonso,vhernand,gmolto}@dsic.upv.es

Abstract

With the advent of Grid technologies, many interest has arisen in the application of these computational techniques to multiple fields. However, Grid Computing technologies have a steep learning curve that tends to discourage scientists from the usage of Grid facilities applied to their research, preventing a widespread adoption of Grid Computing. In this paper, we describe a Java-based middleware, built on top of the Java Commodity Grid, that offers an object oriented, user-friendly view of the Grid, which hides much of the underlying complexity when using the Grid Computing services provided by the Globus Toolkit. The middleware developed is focused on achieving remote execution of tasks, providing automatic file staging services, parallel execution in multiprocessor machines and fault-tolerant scheduling capabilities, from a simple and intuitive Application Programming Interface.

1. Introduction

The introduction of Grid Computing technologies has opened new ways in the computing field by allowing the collaborative usage of computational resources across the world. This situation has enabled to expand the computational barriers of organisations, and specially research centres, by performing time consuming executions on distributed resources from abroad. In fact, the Grid vision [6] aims at delivering computational performance efficiently to a large community of users with such huge computing requirements.

Although there are currently several research projects that aim at developing a middleware that enables to implement that vision, such as Polder [10] or Condor [16], the Globus Toolkit [4], [5] represents the de-facto standard for the deployment of large scale Grid infrastructures.

Unfortunately, users often face a steep learning curve when trying to use the Globus Toolkit features and thus, productivity is reduced and delayed until the toolkit features are mastered. Usually, scientists require the tools to exploit the benefits of the Grid with their own applications, without worrying for the implementation details of the underlying Grid middleware. Therefore, in order to widespread the usage of Grid Computing, this technology must get closer to the end user.

With the purpose of reducing this gap, from Grid technology to the end user, some approaches have been taken into account. The Commodity Grid Kits offer a high level of abstraction that allows computational scientists, who are not expert in Grid programming, to use the Grid services as part of the scientific problem-solving process. In fact, the Java Commodity Grid [17] (CoG) provides, from the Java programming language, an excellent gateway to the Globus Toolkit services, such as GridFTP, the Monitoring and Discovery Service (MDS) or the Globus Resource Allocation Manager (GRAM).

However, a user of the Java CoG has to be aware of the main components and services offered by the Globus Toolkit. Even more, these services must be adequately employed and combined to achieve the intended purpose, which, in general, represents remote task execution. This involves dealing with resource features discovery, file staging, job execution and status control, among others.

In this paper, we present a Java-based software layer, developed on top of the Java CoG, which provides an object oriented view of the Grid, focused on the efficient execution of a set of tasks in a Grid infrastructure. Rather than providing a closed software, we offer an Application Programming Interface (API) that allows the exploitation and even the behaviour modification of the middleware to adapt it, if required, to the computing needs of each specific field.

The article is structured as follows: First of all, section 2 describes the related work, motivating the definition of a new approach. Then, section 3 defines the purpose and intended target of the software layer designed. Next, section 4 introduces the middleware developed, describing the main classes and functionality. Later, section 5 exposes two practical examples of the API usage, as well as its application to port a real simulation system to the Grid. Finally, section 6 concludes the paper, summarising the principal achievements, and pointing out the future plans.

2. Related Work

There are other approaches that provide a framework for simplifying the usage of distributed resources. The Grid Application Framework for Java (GAF4J [11]) is a simple framework of classes that abstracts the essentials of interfacing with a Grid infrastructure, assumed to be Globus Toolkit 2.0. Also, Javelin [14] is a Java-based infrastructure, for Internet-based parallel computing, which provides a framework for achieving execution.

These tools aim at simplifying the process of either porting or developing a Java application to be executed on a distributed environment. However, the middleware being presented allows execution, on Globus-based distributed resources, for already existing applications that can have been written in any programming language, not only Java.

There are also open projects that aim at harnessing the execution on distributed resources. Legion [7] is an object-based middleware that aims at creating a single virtual supercomputer from a set of distributed resources. However, while Legion provides an API for resource management, as well as a for managing executions, it is based on its own self-designed framework, thus being restricted to Legion-based distributed deployments. As another example, NetSolve [3] exposes a self-designed framework in a client-server system that performs fault-tolerant task allocation to computational resources in order to solve complex scientific problems.

One of the advantages of the middleware being presented is that it is compatible with all the largely deployed Globus-based Grids. Examples of these testbeds are the National Technology Grid, that it is being built by the National Partnership for Advanced Computational Infrastructure (NPACI), and the National Computational Science Alliance (NCSA) in order to support distributed scientific and engineering applications, or the LHC Computing Grid Project, which is stablishing a large-scale Grid deployment involving more than 80 sites throughout the world to support application experiments involving real users. Given that Globus is becoming the industrial standard in Grid deployments, compatibility with a broad range of machines is guaranteed.

3. Intended Target of the Middleware

The software layer proposed is designed for the execution of tasks in remote computational resources that form a Grid infrastructure based on the Globus Toolkit. Given that a Grid infrastructure is, at first glance, an unknown pool of computational resources, it can not be assumed that the execution hosts have available the required dynamic libraries that the application depends on, neither the computational nor the system libraries dependences.

Therefore, for Grid execution, the target application must be previously adapted by the user, in order to remove any external dependences, by means of static-linking techniques and disabling all sort of architecture-dependent optimisations [2]. For parallel applications based on the MPI standard, the MPI library can also be introduced into the executable by static linking with a MPICH [8, 9] implementation. These techniques result in a self-contained application, where a minimum level of portability is achieved in order to perform sequential or parallel execution in a wide range of machines that may form the Grid infrastructure.

The middleware designed is focused on the combination of two computational techniques. On the one hand, High Performance Computing enables the speedup of a single task by the parallel execution on a cluster of PCs. On the other hand, High Throughput Computing [13] aims at the effective management and exploitation of all the available computing resources. Therefore, an integration of both techniques, performing parallel executions on the multiprocessor resources of a Grid, seems the key combination to boost productivity.

Our middleware expects a common execution pattern for the applications being ported to the Grid: The application reads a set of input data files, performs a batch computation without interaction with the user, either sequential or in a MPI-based parallel manner, and generates a set of result data files. This is the common behaviour of applications in the field of Computational Fluid Dynamics, Cardiac Simulation, Structural Analysis, and many others, thus covering a wide range of engineering fields.

Therefore, the designed middleware includes a set of components to abstract the process of remote task execution, providing the fundamental tools to perform sequential or parallel executions of a task in a resource. As a result, the user no longer needs to interact with the Globus Toolkit services but with the high level API provided by the implemented middleware.

4. Middleware Description and Components

This software layer has been developed on top of the Java CoG Kit 1.2, which provides a gateway to the functionalities of the Globus Toolkit 2.4 for the Java programming lan-

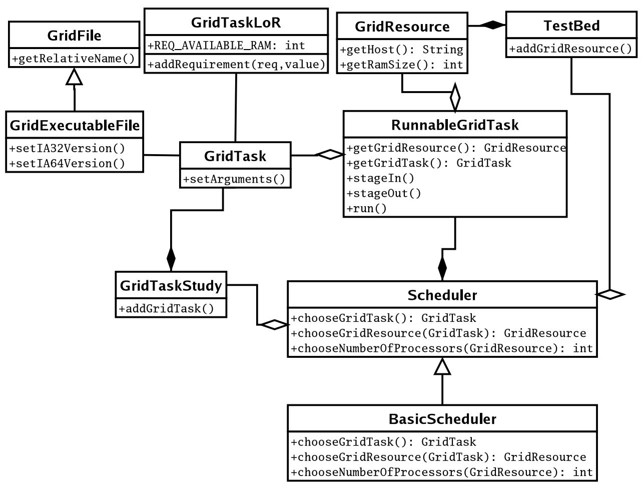

Figure 1. Diagram of the main classes in the middleware.

guage. It must be remarked that a user of the Java CoG Kit needs to know and understand all the different services offered by the Globus Toolkit, and even worse, how to combine them to achieve remote task execution.

For example, porting a scientific application to a Grid deployment requires using several services provided by Globus. First of all, the MDS provides information about the resources. Next, the GridFTP service, or the Global Access to Secondary Storage (GASS) service, enables to achieve data transfer. Finally, the GRAM service allows to achieve execution on the resource. The procedure of remote task execution involves dealing with various interfaces, different specifications and tends to be error-prone due to the inherent complexity of the Globus Toolkit.

Therefore, the idea beyond the middleware presented is to avoid the direct interaction with the Globus services, exposing to the user an object-oriented natural view of the Grid and providing a complete Application Programming Interface for the execution of tasks in resources.

The middleware is composed of more than 30 classes organised in four general packages: Tasks, Resources, Execution and Scheduling, which encapsulate all the functionality related to these four large areas. Fig. 1 summarises the principal and most important classes involved in the middleware. Most of the methods have been omitted for the sake of brevity.

We can see that a TestBed is considered a collection of GridResources and a GridTaskStudy represents a collection of GridTasks. A RunnableGridTask is a GridTask that has been assigned to a GridResource for execution. The Scheduler class defines the interface to all the implemented schedulers which provide capabilities for the allocation of GridTasks to the TestBed.

It should be pointed out that it is responsability of the designed middleware to interact with the Globus services via the Java CoG API. Therefore, the user is provided with an API which no longer exposes the functionality of services such as GridFTP, GRAM or MDS, but a very natural, object-oriented way of interacting with a Grid.

The main classes of the middleware are described in the following sections.

4.1. The GridTask Abstraction

A *GridTask* represents the minimum execution unit in a Grid and has associated several components, such as a *GridExecutableFile* which allows having different executable file versions for the multiple architectures that can be found in an heterogeneous Grid deployment. For example, we may have available an executable file for the IA-32 (Intel, 32 bits) architecture and other for the IA-64 (Intel, 64 bits) platforms, in order to fully exploit the Intel Itanium nodes.

Next, a set of dependent input files can be assigned to a GridTask. They can be specified by an enumeration of files or through a wildcard mechanism. These files will be transferred to the remote host before running the application in order to create an appropriate execution environment.

Analogously, we can also specify the output files that the application will generate upon execution, by means of a wildcard, provided that we may not know the name scheme of the generated archives. Once the execution has finished, these files will be transferred back to the local host in order to access the generated data. A local directory, which will host all these staged out files, can be specified.

4.2. The GridResource Abstraction

A *GridResource* stands as the abstraction for computational resources. This class offers convenient methods to access all the information related to a resource, specially the information supplied by the MDS of the Globus Toolkit.

Therefore, instead of a traditional customized LDAP (Lightweight Directory Access Protocol) query to obtain the required information, the GridResource class offers a collection of methods, such as *getAvailableRam* or *getAvailableProcessors*, which greatly simplify to the end user the process of extracting information from a resource. This information is returned to the user as strongly typed data, thus avoiding the need of data type conversion from the traditional string representation.

For static features of computational resources, such as the operating system, the RAM size or the hardware platform, caching capabilities are included in the middleware to cut down the network usage, therefore reducing the delay caused by multiple consecutive remote LDAP queries. However, when the user requests a dynamic feature, such as the number of available processors, the available RAM or the CPU usage during the last minute, the query is diverted directly to the resource in order to get the most recent information.

4.3. The RunnableGridTask Abstraction

In order to achieve a proper execution of a GridTask in a GridResource, the *RunnableGridTask* offers convenient methods to carry out all the necessary phases.

First of all, the stage in phase must take place, where all the dependent input files of the GridTask and the GridExecutableFile are transferred to the GridResource, using the capabilites of the GridFTP service provided by the Java GridFTP implementation. This process internally involves selecting an appropriate place in the remote filesystem to host the incoming files and to activate the executable flag of the application, which is not set when uploaded. Besides a shell-script, which *wraps* the execution of the application in order to capture its exit code, is also generated and staged in. All this process is achieved with just an invocation to the method `stageIn()`.

In addition, this class provides methods to specify the number of processors that will be involved in the execution, thus allowing to perform a sequential or parallel execution from the same API.

Once the execution environment is appropriate, i.e. after the stage in phase has been completed, the application can be started in the remote GridResource. If the Globus Toolkit installation in the remote resource was properly configured, the execution is integrated with the job manager of the remote GridResource (PBS, LoadLeveler, etc), thus respecting the execution policies of the remote organisation. This process is achieved with just an invocation to the method `run()` offered by this RunnableGridTask.

When the GridTask has finished its execution, the stage out phase is in charge of retrieving all the output files specified, from the remote GridResource to the local machine, storing them in the appropriate local container folder. This is achieved by the method `stageOut()` of this class, which employs the GridFTP service of Globus to accomplish reliable data transfer. Finally, as a polite measure, all the remote temporary files are erased in order to save disk space in the remote resource, as it probably belongs to another organisation.

It is important to point out that all this burden is hidden to the user and so, the middleware is responsible to perform all the mentioned steps to achieve proper execution of the GridTask in the GridResource.

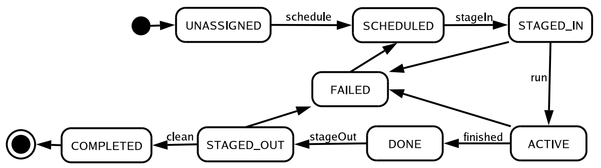

Figure 2. State transition diagram of a RunnableGridTask during its life cycle.

4.4. Introducing A-Priori Quality of Service

A GridTask can optionally specify a list of requirements through the usage of the *GridTaskLoR*. Thus, a GridResource must accomplish these requirements in order to be able to execute the GridTask. This list of requirements currently involve several computational aspects, such as the minimum number of free available nodes in a multiprocessor GridResource, its available MBytes of RAM or the Instruction Set Architecture of the target execution platform.

This way, the GridTask can define an a-priori quality of service, preventing the execution of a resource-starved GridTask on a computationally modest GridResource. This capability is heavily employed during the scheduling process and thus, by fine-tuning the requirements imposed, many computational resources can be discarded as they turn out to be less appropriate than others to host the execution of the GridTask. This strategy tipically leads to shorter execution times and an overall increase in the productivity of the whole task allocation process.

However, it should be considered that a basic knowledge about the behaviour of the application should be known before specifying the requirements. Instrumentation and postmorten [12] analysis are ideal tools to have good estimates about the memory consumption as well as the optimum number of processors for a parallel execution.

4.5. The Scheduler Abstraction

In order to achieve the execution of a GridTaskStudy in a TestBed, a task allocation process must be carried out [15]. We have approached the scheduling process as a state transition of the RunnableGridTasks during their lifetime in the system. This way, the scheduling process is reduced to a state transition problem. Fig. 2 summarises the main states that traverses a RunnableGridTask during its execution.

A RunnableGridTask in the UNASSIGNED state, that is, not yet scheduled for execution, can only be promoted to the SCHEDULED state, by selecting the current best resource. Then, the task is left in the STAGED_IN state by the execution of the method stageIn, which is in charge of performing the stage in phase, as described earlier. Upon starting execution on the remote resource, the RunnableGridTask is transferred to the ACTIVE state. When the execution finishes, the RunnableGridTask usually reaches the DONE state or possible the FAILED state if an error ocurred during execution. After performing the data retrieval phase, its state changes to STAGED_OUT and when the files generated in the remote resource are erased, the RunnableGridTask is considered to be in the COMPLETED state, thus finishing its life cycle.

With this strategy, only a RunnableGridTask that is in a determined state can be promoted to the next one in its life cycle. Failures during the execution of the task on the remote resource, as well as during the data transfer phases, lead the RunnableGridTask to the FAILED state. This condition is detected by the scheduler and mainly involves rescheduling the task, probably to another resource, according to the state transition diagram, until a successful execution is achieved.

The BasicScheduler class provides the implementation of this concept. It is a subclass of the Scheduler class and thus it is comitted to implement the methods which define the policy to select the task (*chooseGridTask*), to select the resource (*chooseGridResource*) and to choose the number of processors of the execution (*chooseNumberOfProcessors*). For each GridTask that must be executed, the BasicScheduler is responsible for its allocation to one resource. Therefore, it selects the current best GridResource, which satisfies the list of requirements of the GridTask, according to the criteria of maximum number of free computing nodes of the GridResource.

This class implements a simple policy for parallel executions. Each of them is limited to employ a quarter the total available processors in the GridResource. Thus, it implements a polite policy that does not involve all the resources in a single execution, and provides the appropriate environment to host up to 4 simultaneous executions, increasing productivity. For resources with up to 4 processors, parallel executions take place with the maximum number of available processors.

The BasicScheduler is a multi-threaded object, where each thread is in charge of the promotion of the RunnableGridTasks from one determined state to the following, according to the state transition diagram. This way, it is possible to concurrently handle an expensive stage in phase of a GridTask while performing the execution of another one, thus reducing the bottleneck of traditional sequential scheduling approaches. Besides the number of threads does not depend on the number of tasks being executed, thus providing a scalable approach in the task allocation process.

Currently, the user has to enumerate the machines required for execution, through the facilities offered by the TestBed abstraction, but plans are to provide support for the GIIS (Grid Index Information Service) functionality of the Globus Toolkit to enable semi-automatic resource discovery.

4.6. Pluggable Scheduling Policies

The design of the middleware allows for an easy implementation of different scheduling policies by reusing the components already designed. Recall the diagram in Fig. 1. The Scheduler class is abstract, providing no implementation but a contract of the interface to be implemented by the different schedulers. For example, it is completely feasible to subclass the BasicScheduler class in order to override the *chooseGridResource* method to implement a different policy for selecting the computational resource in which the GridTask selected by the Scheduler will be executed.

It could also be interesting to override the *chooseGridTask* method in order to prioritise the GridTasks so that those with higher priority (the concept of priority would be defined by the user) are selected first by the scheduler to perform execution.

This subclass of the BasicScheduler class would inherit all the already implemented functionality, what leads to a shorter development time, where the scheduling policies can easily be altered by using the object-oriented functionality of class inheritance as well as method override.

These *pluggable* scheduling policies allow us to test different task allocation schemes under the same Grid deployment (TestBed) in order to measure certain parameters, such as task distribution, total simulation time or idle time of resources, in order to decide the most suitable scheduling strategy for the resources being considered.

4.7. Fault Tolerance Scheme

The GridTask abstraction supports the definition of a checkpoint scheme to introduce execution recovery after failure. Checkpointing is an optional capability of an application which periodically produces a set of files with the current state of the computations, allowing a restarted execution, at a later stage, from the last checkpoint.

Therefore, a GridTask can optionally specify the checkpoint files that the application will generate during its execution, while the scheduler can be configured to automatically and periodically retrieve these files from the remote GridResource to the local machine. Having locally available the latest checkpoint files takes into account failures in the GridResources, allowing to resume the execution on another available computational resource. The applications that do not support checkpointing will be restarted from the beginning upon execution failure.

In fact, when a RunnableGridTask is selected for execution and enters the file stage in phase, it is investigated if there is checkpointing information available, which will be staged to the execution resource. This way, when the application starts the execution on the remote machine, it can be aware of the presence of a checkpoint, thus automatically resuming execution instead of restarting from the beginning. Of course, the application must provide proper support to the generation of its own checkpoint files as well as resuming from its previously generated checkpoint.

5. Practical Examples

Even though all the capabilities of the middleware have been thoroughly explained, it is important to present some examples in order to assess the ease of usage of this approach compared to using the traditional mechanisms provided by Globus.

5.1. Single Remote Execution

A typical situation consists of executing a single application in a remote multiprocessor resource of the Grid. The following extract of Java code will accomplish the execution, employing 4 processors, in a remote resource called *m1*, of an application called *app3D* residing in the */tmp* directory. This task requires a set of input data files, ending in *.dat*. When the execution finishes, all the application-dependent generated data files ending in *.out* will be automatically retrieved. Besides, the GridTask demands that the GridResource has at least 128 MBytes of available RAM. Otherwise the middleware will prevent its execution.

```
GridTask gt = new GridTask();
GridExecutableFile gef =
    new GridExecutableFile("/tmp/app3D");
gt.setGridExecutableFile(gef);
GridInputFileSet gifs =
    new GridInputFileSet();
gifs.addGridFilesBySuffix("/tmp",".dat");
gt.setGridInputFileSet(gifs);
GridOutputFileSet gofs =
    new GridOutputFileSet();
gofs.addGridFilesBySuffix(".out");
gt.setGridOutputFileSet(gofs);
GridTaskLoR gtl = new GridTaskLoR();
gtl.addRequirement(AVAILABLE_RAM, 128);
gt.addGridTaskLoR(gtl);
GridResource gr = new GridResource("m1");
RunnableGridTask rgt =
    new RunnableGridTask(gt,gr);
rgt.setNumberOfProcessors(4);
rgt.stageIn(); rgt.run(); rgt.stageOut();
```

The example uses some classes which do not appear in the main class diagram (Fig. 1). The *GridInputFileSet* encloses the dependent input files of a GridTask, that is, the files that must be accessible in the remote resource prior to execution. The *GridOutputFileSet* indicates the output files that the application will generate during execution, which will be transferred back to the local machine. Notice that a wildcard mechanism is employed to define the files.

For the sake of comparison, using the traditional mechanisms provided by Globus, the user would have to perform the following actions (the parentheses enclose the typical command that would be used to perform the action):

1. To investigate if the resource is alive (*ping*), if the MDS is running (*grid-info-host-search*), if proper credentials are being used to access the resource and if the GRAM service is currently working (*globusrun*).

2. To ensure that the resource has at least 128 MBytes available, by using a customized LDAP query (*grid-info-host-search*) that only returns the value of the attribute *Mds-Memory-Ram-freeMB* by the MDS. Besides, to construct another customized LDAP query that returns the value of the attribute *Mds-Computer-Total-Free-nodeCount*. If there are several execution queues on the resource, to investigate the values for all of them in order to decide if there are at least 4 processors available.

3. To decide where on the filesystem of the remote machine are going to reside all the files and to transfer them via GridFTP or GASS by issuing as many *globus-url-copy* commands as files, as no wildcard mechanism is currently supported. As an additional problem, the GridFTP protocol does not maintain the attributes of the files and thus, the executable file in the remote resource will have the executable flag disabled, thus preventing its execution.

4. To decide if the execution is going to be batch or interactive and to start the execution (*globusrun*) with the number of processors specified. If the execution is batch, then periodically query the status of the job (*globus-job-status*) until the DONE state is reached.

5. To guess where the application will have written its output data files and to transfer them back to the local machine (*globus-url-copy*). To be polite and erase all the generated data files in the resource so that they do not waste disk space in a resource from, possibly, another organisation.

6. To perform error detection along all the previous phases and to take appropriate actions to circumvent them or notify the user.

It can be noticed that, using the traditional mechanisms offered by the Globus Toolkit, a user faces lot of complexity and several design decisions that impose a clear overhead in the process of porting an application to a Grid infrastructure.

The sequence of calls carried out with the proposed middleware involves the usage of the Globus Toolkit services GridFTP, for file transfer, MDS, for resource information, and GRAM, for job execution. However, it is clear that the user is not aware of all this burden, but instead he/she is provided with an API that allows declaring *what* to execute and *where* to execute it, without declaring *how* to achieve execution.

5.2. Multiple Remote Execution

In order to support the execution of a set of tasks, such as those arising in parameter sweep applications, the scheduling capabilities of the middleware can be employed, as shown in the following extract of Java code.

```
GridTaskStudy gts =
    new MyCaseStudy();
String hosts = {"machine1","machine2"};
TestBed tb = new TestBed(hosts);
Scheduler sched =
    new BasicScheduler(tb, gts);
sched.start();
sched.waitUntilFinished();
```

The MyCaseStudy class inherits from GridTaskStudy and represents the problem-dependent definition of the set of GridTasks that have to be executed in the available testbed. This class is supposed to use the API provided by the designed middleware to create the GridTasks, which are included in the study via the *addGridTask* method.

The scheduler will be in charge of the allocation of GridTasks to the *machine1* and *machine2* GridResources, automatically restarting the failed executions or resuming if the GridTask provides checkpointing capabilities. When each GridTask finishes, a folder is created in the local machine, which will host all the output data files collected from the remote GridResource. The invocation of the method *waitUntilFinished* causes the flow control to be stopped while the scheduling process is being performed. Once the task allocation process is finished, the method gives the control back.

It can be noticed from the example that it is very straightforward to employ the scheduling capabilities of the middleware. Just the GridTasks to be executed must be indicated, through the definition of the MyCaseStudy class, and the Globus-based resources that will provide execution services must be specified, through the TestBed class. The task allocation procedure is performed by the middleware.

In the example, it is assumed that the specified resources are running Globus Toolkit 2.4 and valid credentials are installed in the local machine. If, for any reason, no access is available to a resource it will be automatically discarded by the TestBed class so that it will not take part in the scheduling process.

5.3. Porting a Real Application to the Grid

This middleware is currently being employed for the execution of parametric simulations in the field of cardiac electrical activity. These are computationally expensive tasks with an iterative behaviour that reproduce the electrical phenomena of cardiac tissues under a situation of stimulation produced by an electrical shock. Previously, a parallel system for the simulation of action potential propagation on cardiac tissues [1], had been developed. The system accelerated the execution of a single simulation on a cluster of PCs.

However, cardiac case studies such as the analysis of ischemic situations, a condition that may provoke ventricular fibrilation, requires the execution of different parametric simulations where, for example, the degree of the ischemia is altered. The results of each simulation, typically the evolution of the electrical activity of the cardiac tissue, are analysed in order to detect how much distorsion is introduced in the shape of the action potential, the principal indicator of the electrical activity.

This application has been succesfully ported to a Globus-based Grid deployment by using the middleware being described. First of all, the simulator was properly adapted, using static-linking techniques as well as disabling the architecture dependent optimisations in the compilation process, in order to achieve execution on a broad range of Linux machines, one of the principal targets on a Globus-based Grid. A case study which analysed the effects of myocardial ischemia was successfully executed using this middleware. It consisted of 20 independent parametric cardiac simulations, where the time before the ischemic situation took place in the heart was the parameter to vary.

A Java application, using the functionality offered by our middleware was easily developed, creating the IschemiaCaseStudy class, subclassing the GridTaskStudy class, which defined the collection of GridTasks that had to be executed. For each GridTask, we specified executable files for the Intel IA32 and IA64 architectures, in order to achieve good performance on Intel Itanium platforms, as well as the command-line arguments, the input and output archives, the application-dependent checkpoint files that the simulator periodically generates and the rest of capabilities available for a GridTask. A TestBed object was instantiated which encapsulated the computational resources available from universities in Valencia and Madrid (Spain), conforming a Grid deployment with more than 60 processors.

An instance of the BasicScheduler class was created to delegate the process of task allocation in the Grid deployment. Using the capabilities of this middleware we no longer had to use complicated RSL commands in order to achieve remote task execution. We defined *what* to execute and provided all the information required so that the middleware performed the appropriate actions.

6. Conclusions

This paper has presented a middleware which offers an object-oriented approach to Grid Computing technologies, encapsulating part of the functionality provided by the Globus Toolkit in a set of classes that abstract the process of remote task execution.

The middleware offers all the necessary capabilities in order to carry out the execution of tasks in remote resources, performing automatic file staging and parallel execution of MPI-based applications in multiprocessor Grid nodes. Besides, it provides scheduling functionality, performing a fault-tolerant task allocation, based on application-dependent checkpointing or restarted executions, on a Globus Toolkit-based Grid deployment.

This middleware, offered as an API from a Java framework, allows the exploitation of the Globus Toolkit functionality without requiring the knowledge of its provided services. Therefore, reducing the gap from Grid Computing technologies to the end user represents a step forward for the widespread adoption of these computational techniques, what would otherwise be restricted to computer scientists with a knowledge of the Grid middleware services.

Future works involve the extension of the capabilities of the middleware to target other Globus-based Grids such as the middleware being developed in the framework of the european EGEE project, which aims at creating the largest Grid deployment for scientific users. This would enable to expand the range of available machines by several orders of magnitude.

Acknowledgements

The authors wish to thank the financial support received from the Spanish Ministry of Science and Technology to develop the project GRID-IT (TIC2003-01318). This work has been partially supported by the Structural Funds of the European Regional Development Fund (ERDF).

References

[1] J. M. Alonso, J. M. Ferrero (Jr.), V. Hernández, G. Moltó, M. Monserrat, and J. Saiz. High Performance Cardiac Tissue Electrical Activity Simulation on a Parallel Environment. In *Proceedings of the First European HealthGrid Conference*, pages 84–91, January 2003.

[2] J. M. Alonso, V. Hernández, and G. Moltó. Grid Computing Based Simulations of the Electrical Activity of the Heart. In L. N. in Computer Science. Springer-Verlag, editor, *Computational Science - ICCS 2004*, volume 3036. Part I, pages 482–485, 2004.

[3] D. Arnold, S. Agrawal, S. Blackford, J. Dongarra, M. Miller, K. Seymour, K. Sagi, Z. Shi, and S. Vadhiyar. Users' Guide to NetSolve V1.4.1. Innovative Computing Dept. Technical Report ICL-UT-02-05, University of Tennessee, Knoxville, TN, June 2002.

[4] I. Foster and C. Kesselman. Globus: A Metacomputing Infrastructure Toolkit. *Intl. J. Supercomputer Applications*, 11(2):115–128, 1997.

[5] I. Foster, C. Kesselman, and S. Tuecke. The Anatomy of the Grid: Enabling Scalable Virtual Organizations. *International Journal of High Performance Computing Applications*, 15(3):200–222, 2001.

[6] I. Foster and K. Kesselman. *The GRID: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann, 1999.

[7] A. Grimshaw and W. Wulf. The Legion Vision of a Worldwide Virtual Computer. *Communications of the ACM*, 40(1):39–45, 1997.

[8] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A High-Performance, Portable, Implementation of the MPI Message Passing Interface Standard. *Parallel Computing*, 22(6):789–828, 1996.

[9] W. D. Gropp and E. Lusk. *User's Guide for MPICH, a Portable Implementation of MPI*. Mathematics and Computer Science Division, Argonne National Laboratory, 1996.

[10] K. Iskra, R. Belleman, G. van Albada, J. Santoso, P. Sloot, H. Bal, H. Spoelder, and M. Bubak. The Polder Computing Environment: A System for Interactive Distributed Simulation. *Concurrency and Computation: Practice and Experience*, 14(14):1313–1335, 2002.

[11] A. Jhoney, M. Kuchnal, and S. Venkatakrishnan. Grid Application Framework for Java (GAF4J). A Technical Whitepaper. http://www.alphaworks.ibm.com/tech/GAF4J.

[12] S. Krishnan and L. V. Kale. Automating Parallel Runtime Optimizations Using Post-Mortem Analysis. In *International Conference on Supercomputing*, pages 221–228, 1996.

[13] M. Livny, J. Basney, R. Raman, and T. Tannenbaum. Mechanisms for High Throughput Computing. *SPEEDUP Journal*, 11(1), June 1997.

[14] M. Neary, S. Brydon, P. Kmiec, S. Rollins, and P. Cappelo. Javelin++: Scalability Issues in Global Computing. *Concurrency: Practice and Experience*, 12:727–753, 2001.

[15] J. M. Schopf. Ten Actions When Superscheduling. SchedWD 8.5, Scheduling Working Group, 2001.

[16] D. Thain, D. Wright, K. Miller, and M. Livny. Condor - A Distributed Job Scheduler. *Beowulf Cluster Computing With Linux*, 2001.

[17] G. von Laszewski, I. Foster, J. Gawor, and P. Lane. A Java Commodity Grid Kit. *Concurrency and Computation: Practice and Experience*, 13(8-9):643–662, 2001.