Grid Enabled JaSkel Skeletons with GMarte

J.M. Alonso[1], V. Hernández[1], G. Moltó[1], A. Proença[2], and J.L. Sobral[2]

[1] Universidad Politécnica de Valencia, Departamento de Sistemas Informáticos y Computación, 46022, Valencia, Spain
[2] Universidade do Minho, Departamento de Informática, 4710-057, Braga, Portugal

Abstract. The development of parallel applications requires adequate tools to help the programmer to create applications that can efficiently execute on a wide range of computing platforms. This paper describes a joint effort to build a programming framework that can transparently execute parallel applications on multiprocessor systems, clusters and computational Grids. This environment integrates a Java skeleton-based framework, JaSkel, and a Java-based Grid middleware, GMarte, to transparently execute skeletons across cluster boundaries. The resulting framework allows JaSkel applications to seamlessly use computational Grids. This joint software leverages both previous independent efforts to provide easier access to computational resources.

1 Introduction

The development tools for parallel computing must address a variety of computing architectures, from multi-core to Grid systems, and must take advantage of the multi-level nature of computational Grids such as a federation of clusters with multi-core and shared-memory computing nodes. Programmers do not want to be aware of these details and prefer to focus on the computational side of their (domain-specific) codes. Programming frameworks should enable a more seamless access to computational resources, providing efficient execution of parallel codes on a wide range of computing platforms. Current frameworks for parallel computing address either cluster or Grid environments, but not both.

JaSkel [8] is a skeleton framework, developed in Portugal, which helps the programmer to build well structured parallel applications, by providing a set of common parallelisation patterns as Java skeletons. In JaSkel a programmer builds a parallel application by selecting (and composing) parallel patterns that implement its structure and fills the skeleton with domain specific code. With the current JaSkel system, programmers can write code that efficiently runs on shared-memory systems and clusters.

The GMarte [5] framework, developed in Spain, is a Java-based Grid middleware that simplifies the process of remote task execution on computational Grids, exposing an object-oriented application programmer interface. Using GMarte the programmer can develop Grid enabled applications without being aware of the details of the underlying Grid middleware, such as the Globus Toolkit [9].

This paper presents these separate software tools and describes preliminary efforts to develop an integrated parallel computing framework that addresses

both the execution of parallel applications on clusters and Grid environments. It describes how JaSkel skeletons can be transparently run on Grid platforms using the GMarte middleware. This integrated environment allows a transparent migration of current JaSkel applications to Grid infrastructures.

This paper is organised as follows. Section 2 and 3 present the GMarte middleware and the JaSkel framework, respectively. Section 4 describes the integration of these two frameworks into a single environment. Section 5 gives an overview of the state of the art in this area and Section 6 concludes the paper.

2 The GMarte Framework

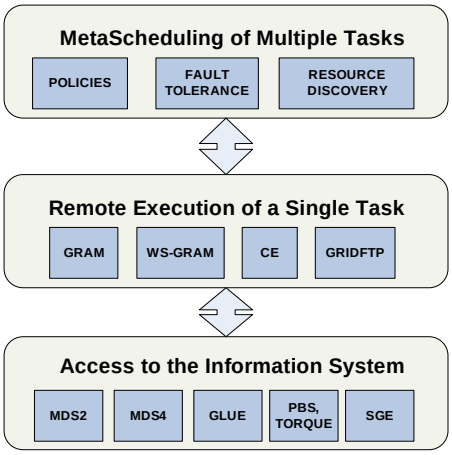

Fig. 1. Layered diagram with the abstraction framework provided by GMarte.

GMarte [4, 5] is a software framework, which aims to simplify the process of executing scientific applications on computational Grids based on the standard Globus Toolkit [9, 10].

This software has been developed as a Java library, exposing a high level API (Application Programming Interface) to interact with a computational Grid. The choice of Java responds to a two-fold strategy. First of all, its portability enables to seamlessly use the GMarte functionality from virtually any platform. In addition, its object-oriented approach eases extensibility, for program developers, and exposes high-level object abstractions for the integration of its functionality in other scientific or engineering applications.

Figure 1 shows the principal abstraction layers available in the GMarte framework. On the bottom, it first provides an homogeneous interface to access the Information Systems of the heterogeneous computational resources. Over this common API, a layer which enables remote task execution on Globus-based resources was implemented. Later, the capabilities of GMarte were extended by

introducing metascheduling functionality for the allocation of multiple tasks on computational Grids.

Figure 2 summarises the principal high-level object abstractions available in GMarte to interact with computational Grids. The *GridTask* and *GridResource* objects enable to define a task and a computational machine, respectively. A *Scheduler* object provides meta-scheduling functionality for the allocation of a *GridTaskStudy* (a set of independent tasks) to a *TestBed* (a set of machines). The following sections summarise the principal functionality of GMarte.

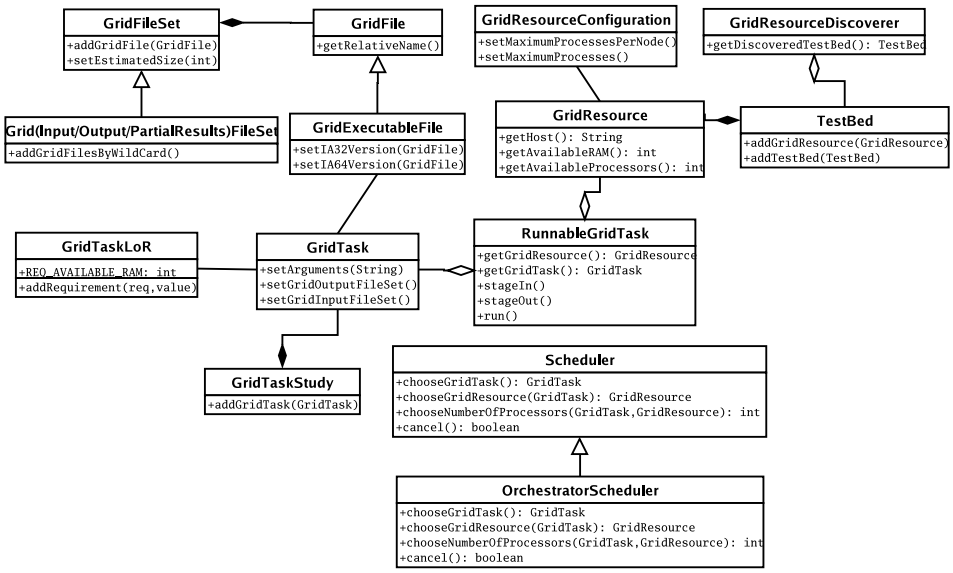

Fig. 2. UML diagram of the principal abstraction framework provided by GMarte.

2.1 Homogeneous Access to the Information System of The Computational Resources

The Monitoring and Discovery (MDS) service publishes the computational information of resources. However, this information is exposed via different protocols and data formats, depending on the Grid middleware. To hide these differences, GMarte provides an abstraction layer to access these data. For that, a high level object, called *GridResource*, offers a common API with user-level methods (getAvailableProcessors, getAvailableRAM, etc.) that gather the requested information, in a transparent way for the user. This is achieved by using the appropriate protocols depending on the Grid middleware of the remote machine.

At the moment, these features in GMarte are available for resources with the Globus Toolkit 2.X and 4.X, as well as the Computing Elements in LCG-2[3] deployments. The supported LRMS is PBS/Torque (Portable Batch System).

2.2 Abstracting the Process of Remote Task Execution

GMarte abstracts the process of remote task execution by performing fault-tolerant executions with coordinated data staging, even for parallel applications based on the MPI (Message-Passing Interface) library.

The *GridTask* object provides the abstraction to define the computational task to be executed on the Grid. It allows specifying the dependent input files as well as the output ones that generates. In addition, special execution requirements for the task can be set, such as the minimum available RAM of a resource to execute it. This information will be considered by the metascheduler to filter out machines that do not satisfy these requirements.

First of all, the application and its dependent input files are transferred via GridFTP from the client to the remote machine (a process called *stage-in*). Then, the application is started in the remote machine via the GRAM (or WS-GRAM) service and it is periodically monitored to detect possible failures. When the execution finishes, the output files generated are transferred to the client machine via GridFTP (a phase called *stage-out*).

In this layer, GMarte implements a multi-level fault tolerance scheme which allows to recover itself from failures both during the data transfers (which are retried a certain amount of times before giving up), as well as during task execution, where failed tasks will be marked so that the metascheduler can re-schedule them. Failures in remote computational resources also cause a re-scheduling of the tasks on other available machines.

2.3 Metascheduling Approach

The metascheduling procedure available in GMarte involves several phases, as shown in Figure 3. First of all, the *Resource Discovery* phase involves the discovery of a set of candidate execution machines from either a GIIS (Grid Index Information Service) or a BDII (Berkeley Database Information Index). These components aggregate resource information and are queried to provide a list of machines.

Once a list of potential execution machines have been obtained, the *Resource Filtering* phase is in charge of discarding those resources that either are not accessible with the user credentials or do not satisfy the computational requirements of the task to be executed.

When both previous phases finish, a set of computational resources on which to perform the execution of the tasks is available. In order to decide which resource is going to run each task, a *Resource Selection* stage must take place for each of them, which chooses the current best machine according to some

[3] LCG - LHC Computing Grid : http://www.cern.ch/lcg

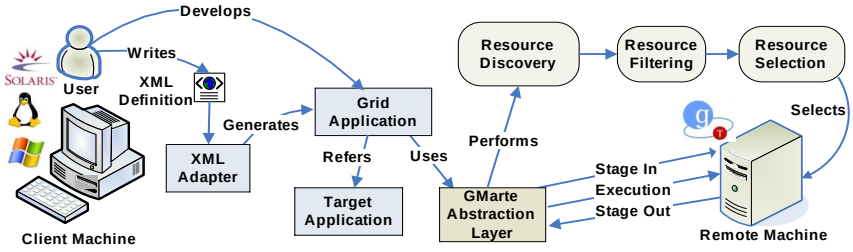

Fig. 3. Overview of the principal phases involved in the GMarte framework.

criteria. GMarte implements different resource selection policies and, being an abstraction framework, the user can easily create new policies by providing different score functions for computational resources. As no resource selection can be considered the best for all kind of applications, this allows to extend the functionality via additional user-programmed modules.

GMarte implements a multi-threaded metascheduler that enables to concurrently perform the distinct phases involved in the remote task execution of different tasks (resource selection, stage-in, execution and stage-out). For example, the user can specify the number of threads involved in the stage-in phase. If 3 threads are specified, then up to 3 tasks will be simultaneously handled, thus enabling to reduce the overhead introduced by the metascheduler.

Finally, it should be mentioned that the functionality described can be used via a GMarte API-based Java application as well as through XML (eXtensible Markup Language) documents, which are automatically mapped to the corresponding abstraction objects.

Currently the GMarte framework has been successfully applied for the execution of a biomedical application that simulates the cardiac electrical activity both on local and large-scale Grid infrastructures [2]. Its functionality has also been introduced as part of a Grid service that performs the structural analysis of buildings [3]. This has enabled to perform executions of scientific applications in a transparent manner for the user.

3 The JaSkel Framework

Skeleton-based frameworks provide a set of pre-defined structures (called skeletons) to develop parallel applications. The programmer selects the skeletons that best suit his/her concrete application, provides application specific code and the framework takes care of concurrency and distribution issues. By using skeletons the programmer can focus on the computational side of their algorithms rather than on parallelisation issues.

Current JaSkel framework [8] provides several skeletons, including pipeline, farm and heartbeat. Each of these skeletons implements a frequently occurring structure of parallel applications. To write an application in JaSkel the programmer must go through three steps:

1. Select the skeleton(s) that better fits the application parallelisation.
2. Refine the skeleton abstract class, providing the domain specific code, by filling skeleton hooks (i.e., abstract methods).
3. Write code to instantiate the skeletons and start the skeleton activity.

A JaSkel skeleton is a Java class that extends the *Compute* class (Figure 4). The skeleton method *eval* starts the skeleton activity.

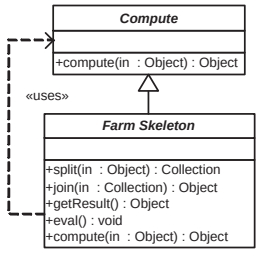

Fig. 4. The Farm Skeleton.

Compute objects perform domain-specific computation. The programmer must create a subclass of class *Compute*, implementing the abstract method *Object compute(Object input)* to provide domain-specific computations.

To create a parallel application based on a farm parallel structure, the programmer must follow these steps:

1. Create the worker *Compute* class;
2. Create the farmer class, extending *Farm*, and implementing the *split* and *join* methods.
3. Create instances of worker and farmer class to achieve the intended farm structure;
4. Start the skeleton activity, by calling the *eval* and grab the results with the *getResult* method.

When the *eval* method is called, it performs the following steps:

1. Split the initial data using the farmer *split* method.
2. Call the *compute* method from workers with the pieces of data returned by the *split* method.
3. Merge the partial results using the farmer *join* method.

JaSkel skeletons are also subclasses of *Compute* and implement the *compute* method, as shown in Figure 4, to support skeleton nestings.

The following code presents a simple pseudo-code for a farm. Lines 01-06 define the Worker class, implementing the *compute* method. Lines 07-15 implement a Farm class that creates its own set of Worker and provides specific split and join methods. Lines 17-21 instantiate the Farm, start the skeleton activity and get the computed task.

```
01  public class Worker extends Compute {
02
03      public Object compute(Object input) {
04        return /* processed input */;
05      }
06  }
07  public class MyFarm extends Farm {
08
09    public MyFarm() {
10      for(int i=0; i<numberWorkers; i++)
11          /* ... */ = new Worker();
12    }
13    Collection split(Object initialTask) { ... }
14    Object join(Collection partialResults) { ... }
15  }
16
17  // main function
18  Task task = ... // task to compute
19  MyFarm farmer = new MyFarm(task);
20  farmer.eval(); // starts the farming process
21  Object result = farmer.getResult(); // get results
```

4 Jaskel-GMarte Binding

JaSkel has built-in support for concurrency, through several concurrent skeletons. For instance, the *ConcurrentFarm* skeleton provides concurrent execution of the *Compute* instances. These skeletons with built-in concurrency support can take advantage of shared memory systems, as they rely on a thread model with embedded support for data sharing among threads. However, these skeletons can not take advantage of distributed memory systems.

Support for skeleton execution on distributed memory systems is provided by external tools. These tools transform concurrent skeletons into other ones that can be executed in remote nodes. This section describes the developed distribution tool that can transparently execute JaSkel Farm skeletons on computational Grids through GMarte. It starts to show how Java applications can be executed using GMarte, describes the JaSkel-GMarte binding and presents preliminary performance results.

4.1 GMarte Java API

To support the execution of Java-based applications, enhancements have been introduced in GMarte. These executions rely on a JVM (Java Virtual Machine) in the remote computing node, which requires a remote search for a JVM prior to task execution, since the Java location is not commonly published in the MDS.

The easiest way to remotely execute a Java application is to pack its contents into a JAR (Java Archive) file. This ensures that all the application dependences will be available when the execution starts. The following example shows how to use the GMarte functionality to execute a Java-based application packed into the *myapp.jar* file. This toy application takes as input the file *schema.xsd* and generates a set of **.java* files. The command-line executed to run the application is *java -jar myapp.jar schema.xsd*.

```
GridTask gt = new GridTask("My Java Task");
gt.setGridExecutableFile(new JavaGridExecutableFile());
gt.setLocalDataContainer("/home/user/outputData");
gt.setArguments("-jar myapp.jar schema.xsd");
GridInputFileSet gifs = new GridInputFileSet();
gifs.addGridFile("/home/user/schema.xsd");
gt.setGridInputFileSet(gifs);
GridOutputFileSet gofs = new GridOutputFileSet();
gofs.addWildCard("*.java");
gt.setGridOutputFileSet(gofs);
GridTaskStudy gts = new GridTaskStudy();
gts.addGridTask(gt);
GridResource gr = new GridResource("amachine.lab.com");
TestBed tb = new TestBed();
tb.addGridResource(gr);
Scheduler scheduler = new OrchestratorScheduler(tb, gts);
scheduler.start();
scheduler.waitUntilFinished();
```

The programmer uses high-level object abstractions to define the computational task. Notice that the definition of the computational resources is provided via the TestBed object. Alternatively, the user may have relied on a resource discovery process. Also notice that the GridTask is enclosed in a GridTaskStudy object. This enables the submition of a group of independent tasks. With this abstraction framework, the user no longer interacts with the lower level details of the Grid but instead focuses on the scientific side without detailing how the executions will be performed.

4.2 GMarte-based Distributed JaSkel Farm Skeletons

The JaSkel distribution tools are based on aspect-oriented techniques [13] to transparently replace user-provided skeletons with new implementations. The idea is to generate a new type-compatible class that transparently replaces the original skeleton. This new type-compatible skeleton encapsulates the access to remote skeleton instances. Moreover, it is possible to plug and unplug this new implementation [16] at compile or load-time, providing an easy way to plug or unplug this behaviour from the original JaSkel code. For instance, in the example

of section 3, the tool would generate a new *Worker* class (i.e., worker proxy) that instantiates the worker on a remote resource and overrides the *compute* method to redirect the execution to the remote resource. Figure 5 illustrates this process.

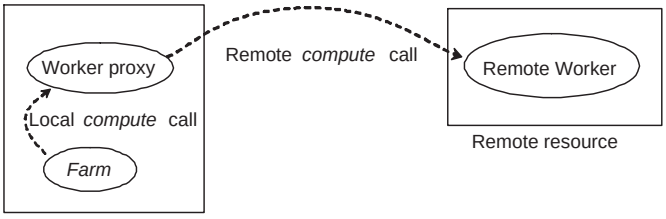

Fig. 5. Remote skeleton.

The distribution tool identifies classes that extend the *Compute* class and generates two modules for each class: the client side and the server side code. The client-side module replaces source code references to *Compute* skeletons by the corresponding proxy skeleton. The server side code transforms a *Compute* instance into a stand-alone task that can be executed on a remote resource and that accepts remote requests for method execution. Remote requests are identical to the *Remote Procedure Call* SOAP message pattern. However, the generated code uses a proprietary, more efficient, wire protocol, since interoperability is not a concern in this case, as both client and server are generated by the distribution tool. The next two sub-sections describe in more detail each of these modules.

Client Side Code. The client side code is in charge of redirecting the execution of *compute* methods to remote resources. The client-side worker proxy overrides the *compute* method to produce a request to a remote skeleton. It writes the request to a file that is *staged-in* to the remote resource. The GMarte framework is responsible to *stage-in* this file and it is also responsible to copy the code of the remote worker (e.g., a jar file), to start the remote worker and to *stage-out* the resulting file. The following code illustrates the code generated for the proxy.

```
public class GridWorker extends Worker {  // worker proxy
  public Object compute(Object obj) {
    ...  // generate RPC request (plain text file)
    GridTask gt = new GridTask("JaSkel worker);
    GridInputFileSet gifs = new GridInputFileSet();
    gifs.addGridFile("jaskel.jar"); // remote skeleton jar file
    gifs.addGridFile(/* RPC request file name */);
    gt.setGridInputFileSet(gifs);
    GridOutputFileSet gofs = new GridOutputFileSet();
    gofs.addGridFile(/* result file name */);
    gt.setGridOutputFileSet(gofs);
```

```
    ...  // request task execution to GMarte (see section 4.1)
    ...  // read result from staged-out file
    return(/*result*/);
  }
}
```

Note that the remote execution was completely abstracted by the GMarte Java API, making the generated code independent of the Grid middleware.

Server Side Code. The code of the remote worker creates a stand-alone skeleton (e.g., *Worker* instance) that reads the file containing the RPC request, executes the requested method and writes a file with the result of the *compute* call. The GMarte framework is responsible to copy the required input and output files to the resource where the skeleton is executed. However, the code of the remote worker does not depend of the GMarte framework. The following code illustrates the code of the remote worker.

```
public static void main(String args[]) { // remote task to execute
  Worker myWorker = new Worker();
  Object myData = ... // read data from file staged in by GMarte
  Object result = myWorker.compute(myData);
  ... // save result into output file to be staged out by GMarte
}
```

4.3 Performance Results

Table 1 shows execution times of the Raytracer from [8] (4000x4000 image) on a computational Grid made of two clusters. The table also presents running times of the original JaSkel application on a 4-CPU, MacPro Xeon 5130 system. This version provides the lowest execution time for 1 to 4-CPUs, however it cannot take advantage of computational Grids. The integrated framework provides lower running times by using more CPUs, but it does not scale beyond 16 CPUs due to the remote task execution overheads (e.g., job stage-in and stage-out).

Table 1. Raytracer Execution Times (in seconds)

CPUs	1	2	4	8	16	32	50
Grid	931	497	285	187	143	156	167
MacPro	730	363	202				

5 Related Work

Software tools that aim to simplify the process of task execution of Java applications in a Grid infrastructure include: the Grid Application Framework for Java (GAF4J) [12] abstracts the interaction with a Grid platform based on Globus 2.0, while ProActive [6] is a Java library for concurrent, parallel and distributed computations, which creates distributed active objects. However, GMarte can also be employed for applications developed in other programming languages.

Metascheduling on computational Grids based on the Globus Toolkit are addressed by GridWay [14], an open source metascheduler that provides reliable unattended execution of jobs on computational resources. However, it runs on Unix-based systems, while GMarte can be run on other platforms.

Several skeleton based approaches are described in the literature [15, 7]. Lithium [1] and CO2P3S [17] are two Java-based skeleton environments for parallel programming. HOCs [11] were specifically designed for Grid environments by supporting independent deployment of skeleton and application specific code.

JaSkel differs from these skeleton systems by relying on tools to adapt skeletons to specific distributed environments. This approach enables skeletons to scale to a wide range of computing platforms, from multi-core systems to computational grids and allow JaSkel skeletons to combine multiple communication middleware into a single application, taking advantage of the layered composition of clusters and Grids.

6 Conclusions

This paper explored the binding between a skeleton-based framework and a Grid middleware. The integrated environment helps developers to run applications on a wide range of computing platforms, without changing a single line of code. This novel programming framework supports the development of HPC applications that can efficiently run on multi-core systems, clusters and Grids.

JaSkel framework aims to support multiple middleware by relying on external tools to generate distribution code, while GMarte provides a Java based API to simplify the task execution on computational Grids. The following goals have been achieved:

- *JaSkel framework was able to accommodate a new distribution middleware*: the JaSkel-GMarte binding did not required any change to the internal structure of JaSkel skeletons, it just required a new distribution tool.
- *GMarte simplified the code required to execute skeletons on computational Grids*: JaSkel execution on top of GMarte is supported by its Java API. The code required for this task is small when compared to *de facto* Grid middleware and is loosely dependent on Grid related issues.

Current work includes a tighter integration of JaSkel skeletons and the metascheduling facility of GMarte, for instance, to compute the ideal number of skeletons to deploy or to schedule parallel tasks on remote nodes (e.g., sets of *Compute* instances), reducing the stage-in and stage-out time.

Acknowledgments

This work was supported by PPC-VM project (Portable Parallel Computing Based on Virtual Machines, POSI/CHS/47158/2002) and by SeARCH (Services & Advanced Computing with HTC/HPC, CONC-REEQ/443/EEI/2005) both funded by Portuguese FCT (POSI) and European funds (FEDER).

References

1. Aldinucci, M., Danelutto, M., Teti, P.: An advanced environment supporting structured parallel programming. Java, FGCS. **19**, (2003).
2. Alonso, J.M., Ferrero (Jr.), J.M., Hernández, V., Moltó, G., Monserrat, M., Saiz, J.: Three-dimensional cardiac electrical activity simulation on cluster and Grid platforms. Lecture Notes in Computer Science **3402** (2005) 219–232.
3. Alonso, J.M., Hernández, V., López, R., Moltó, G.: A service oriented system for on demand dynamic structural analysis over computational Grids. Lecture Notes in Computer Science **4395** (2007) 13–26.
4. Alonso, J.M., Hernández, V., Moltó, G.: An object-oriented view of Grid computing technologies to abstract remote task execution. Proceedings of the Euromicro 2005 International Conference, (2005) 235–242.
5. Alonso, J.M., Hernández, V., Moltó, G.: GMarte: Grid middleware to abstract remote task execution. Concurrency and Computation: Practice and Experience **18** (15) (2006) 2021–2036.
6. Baude F., Baduel L., Caromel D., Contes A., Huet F., Morel M., Quilici R.: Programming, composing, deploying for the Grid. Grid Computing: Software Environments and Tools, J. Cunha & O. Rana (Eds), Springer Verlag, January 2006.
7. Cole, M.: Bringing skeletons out of the closet: a pragmatic manifesto for skeletal parallel programming. Parallel Computing, **30** (3), (2004).
8. Fernando, J., Sobral, J., Proença, A.: JaSkel: A Java skeleton-based framework for structured cluster and Grid computing. IEEE CCGrid'06, Singapore, (2006).
9. Foster, I., Kesselman, C.: Globus: A metacomputing infrastructure toolkit. Intl. J. Supercomputer Applications **11** (2) (1997) 115–128.
10. Foster, I.: Globus toolkit version 4: Software for service-oriented systems. IFIP Int. Conference on Network and Parallel Computing, Vol. 3779, (2005) 2–13.
11. Gorlatch, J., Dunnweber, J.: From Grid middleware to Grid applications: Bridging the gap with HOCs. Future Generation Grids, Springer (2006).
12. Jhoney, A., Kuchhal, M., Venkatakrishnan, S.: Grid application framework for Java (GAF4J). http://dl.alphaworks.ibm.com/technologies/gaf4j/GAF4Jwhitepaper1.2.doc
13. Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C. V., Loingtier, J.-M., Irwin, J.: Aspect Oriented Programming, ECOOP'97, Finnland, (1997).
14. Montero, R.S., Huedo, E., Llorente, I.M.: Grid resource selection for oportunistic job migration. Lecture Notes in Computer Science. **2790**(1) 366–373.
15. Rabhi, F., Gorlatch, S.: Patterns and skeletons for parallel and distributed computing, Springer (2003).
16. Sobral, J.L., Cunha, C., Monteiro, M.: Aspect oriented pluggable support for parallel computing. Lecture Notes in Computer Science **4395** (2007) 93–106.
17. Tan, K., Szafron, D., J. Schaeffer, J., Anvik, J., MacDonald, S.: Using generative design patterns to generate parallel code for a distributed memory environment. ACM PPoPP'03, San Diego, California, USA (2003).