Non-Invasive Gridification through an Aspect-Oriented Approach

Edgar Sousa, Rui Carlos Gonçalves, Diogo Neves, João Luís Sobral

Centro de Ciências e Tecnologias da Computação
Departamento de Informática
Universidade do Minho, Braga, Portugal
{edgar, rgoncalves, dneves, jls}@di.uminho.pt

Abstract. This paper presents a framework that allows plugging or unplugging Grid-related features to legacy code. Those features can, for example, transform sequential-like code to a parallel version or even a distributed version. With this approach scientists could develop their code as they are used to, that is, they could spent their time programming as they are used to instead of learning new programming paradigms. The framework relies on aspect-oriented programming (AOP) techniques to perform non-invasive enhancements to scientific applications to cope with grid specific issues, such as remote execution, load distribution, fault tolerance and monitoring. In addition, framework components are glued together with AOP, thus it is possible to plug (only) features required for each specific application / target platform.

Keywords: grid-enabled applications, grid frameworks, aspect-oriented programming

1 Introduction

Computational grids allow scientists to access to an almost unlimited computing power, making it possible to perform new range of experiments not possible before. Early grid systems targeted parameter sweep applications, where some scientific code is executed many times, providing a different parameter for each execution. For this purpose, most grid middleware rely on a command line interface where the user can specify parameters for each run. For this purpose, most grid middleware relies on a command line interface or on a JDL - Job Description Language [3] - where the user can specify parameters for each run.

Grid-enabling scientific codes that do not rely on parameter sweep require an additional burden to decompose the application into a set of independent tasks that can be executed on a set of remote computing nodes. Moreover, parameter sweep may not be affordable for some kind of applications as some run(s) may dependent on another(s).

Current tools for gridification [14] require invasive and non-reversible modifications to scientific codes (e.g., to specify parallel tasks and to execute these tasks on a grid environment) making grid-enabled codes dependent of a grid infra-structure.

In addition, decomposing an application into a large set of independent tasks and submitting these tasks to a grid scheduler avoids application specific optimisations, such as performing application-level self-scheduling on available computing resources or fault tolerant execution of specific application tasks. Moreover, current tools focus only on executing applications solely on grids, missing to take advantage of the growing number of cores on every computing element.

The AspectGrid framework [1] aims at gridifying scientific applications in a non-invasive manner, through a set of pluggable components [16] that can be composed to meet the requirements of each application/target platform. Current pluggable services include parallelisation, load-distribution, fault-tolerance, remote execution and monitoring.

The rest of the paper is organised as follows. Section 2 overviews the AspectGrid framework, describes its main components and their inter-connection. Section 3 presents performance results and section 4 compares this work against other approaches. Section 5 concludes the paper and points directions for future research.

2 AspectGrid Framework

The AspectGrid framework aims to be a lightweight framework for non-invasive gridification of scientific applications. It relies on a minimalist interface among components that can be extended to match application specific needs, through a set of pluggable components. To achieve this goal the framework strongly relies on AOP techniques. Figure 1 presents an overview of the framework architecture.

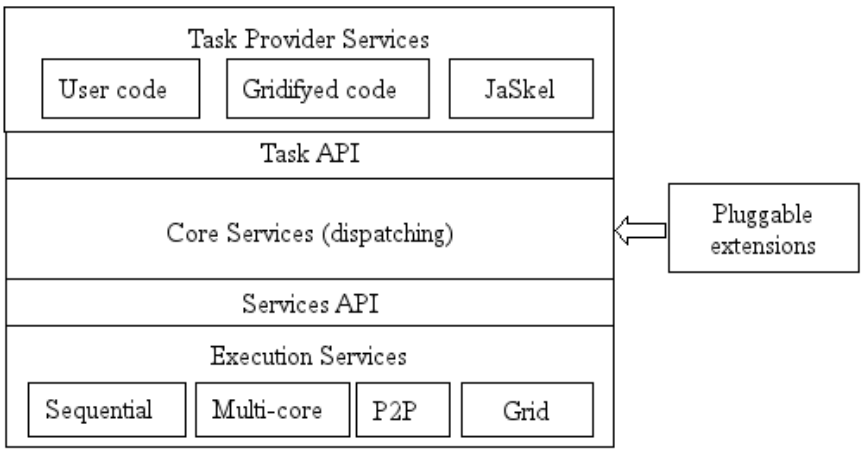

Fig. 1. Ideal framework layout.

Task provider services are domain specific code that generates tasks to be executed on computational resources. Examples of these are scientific applications

gridified by our approach, but it is also possible to use other task providers, for instance by directly (i.e., invasively) creating tasks within the code, or by using a skeleton framework [17]. These tasks can be executed on a set of resources provided by execution services. The core of the framework dispatches tasks to available execution services. This dispatching service can be extended with pluggable features to meet specific application need(s).

The AspectGrid framework, which currently instantiates this architecture, provides pluggable services for parallelisation, remote execution, scheduling, load-distribution, fault-tolerance and monitoring (Fig. 2).

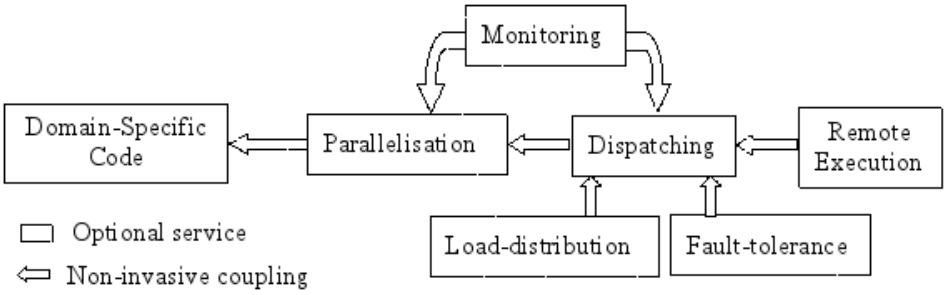

Fig. 2. Main components of the AspectGrid framework.

The parallelisation service is responsible for the generation of a set of independent tasks that can be scheduled into a set of local or remote resources, by the dispatching service.

The load-distribution service performs a more fine-tuned resource selection, based on the specificities of each resource. The fault-tolerance addresses faulty resources by resubmitting a task for execution when its execution fails. Both the load-distribution and fault-tolerance services can be seen as improvements to the basic dispatcher service.

Remote execution service manages task execution on remotes nodes, including code deployment. Finally, the monitoring service can manage the progress of tasks execution.

The key issue in the AspectGrid framework is the ability to bind these services into scientific codes in a non-invasive manner, making it possible to (un)plug these services into/from scientific codes at any time during and after the gridification process. In addition, connections among services are also performed in a non-invasive manner, minimising coupling among services. This approach makes it feasible to develop grid-enabled applications that do not depend on a particular: (1) set of services, as services can be plugged only at request to meet specific execution requirements; (2) target platform, as the code can be tuned to meet (some) specific hardware configurations, such as multicore machines. For instance, to run a scientific application on a local multicore machine, remote execution

and fault-tolerance services are not required. This also presents a performance advantage, since services can be removed from the build.

Non-invasive composition of services relies on aspect-oriented programming techniques [12]. Two fundamental concepts of AOP are quantification and obliviousness [8]. Quantification is the ability of an aspect (or service in our framework) to specify a set of execution points where aspect specific behaviour can be attached. For instance, a service for remote execution can attach that behaviour to certain procedure calls in scientific code. Obliviousness is the ability to apply a mechanism to code that was not specifically prepared for that purpose. To illustrate these two concepts, consider a service that would print the name of every method called. The following pseudo code specifies that we want to apply this aspect to all method calls (`pointcut events2print()`) and print the method name before the method call (`before() : events2print()`, the *joinPoint* construct was used in this case to specify the name of each intercepted execution point).

Algorithm 1 Example of an aspect.

```
pointcut events2print() : call(* *.*(..));

before() : events2print() {
  System.out.println(joinPoint.methodName());
}
```

Both quantification and obliviousness are present in this example. First, the set of method calls to intercept are specified in the pointcut construct (quantification). Second, no special adaptations are required to the basic code to attach this feature (obliviousness).

Fig. 3 presents a more detailed version of the building blocks of the AspectGrid framework. Three additional components have the purpose to provide interfaces among components that are inter-connected by AOP: the *FrameworkAdapter*, the *ITask* and the *IServer* interfaces.

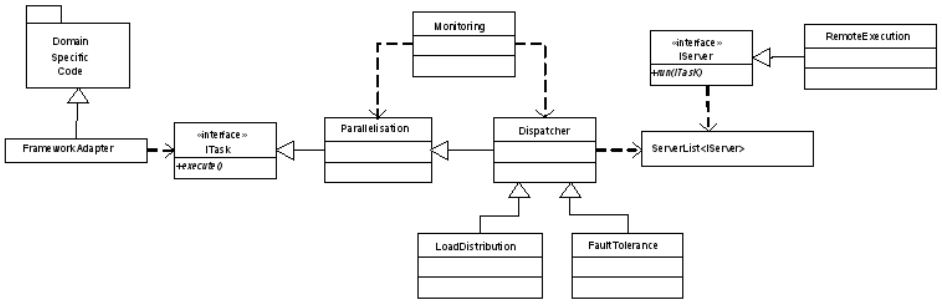

Fig. 3. Glue and components of the AspectGrid framework.

The *FrameworkAdapter* is an application-specific component, automatically generated by the framework that specifies a task to parallelise and/or to execute remotely. Its purpose is to transform a procedure/method call in the scientific code into an *ITask* that can be processed by other framework services. The adapter non-invasively decouples the remainder framework services from the specificities of the domain specific code, by encapsulating tasks into a well defined interface (a strategy similar to the command pattern [9]). User specified method calls are encapsulated into a class implementing the *ITask* interface, where calls to the original method are made inside the method execute. Other framework components (e.g., parallelisation and scheduler) may intercept executions of method execute to plug their services. The parallelisation service can decompose an *ITask* into multiple *ITask*'s using a default or a user provided partition strategy. When no other services are plugged, the default *ITask* implementation (execute method) performs the execution sequential and locally.

The dispatcher assigns *ITask*'s to available resources through the *IServer* interface. For this purpose it uses a list of available *IServer*'s and dispatches an *ITask* by calling the `IServer.run(ITask)` method. The default scheduling policy performs a round-robin assignment of tasks. The default implementation of the *IServer* (e.g., when no remote execution is plugged) directly executes the method by means of a local thread pool (which may take advantage of multi-core machines).

The next subsections describe in more detail the services currently provided by the framework (i.e., parallelisation, scheduling, remote execution) and overviews their current implementation using AspectJ [8], an AOP extension to Java.

2.1 Parallelisation

The parallelisation service acts upon a class implementindg the *ITask* interface and generates a new set of tasks that are executed in parallel. For this purpose two application specific methods should be provided: *scatter* and *reduce*. The *scatter* method specifies how the parameters of the original task are scattered among a set of new tasks. For instance, when processing an array of data, this method can create several smaller arrays of data. In a simulation, it can generate different parameters for each task. The *reduce* method combines the results of each computed task into a single one. In the case of an array of data it can merge all parts of the processed data.

2.2 Dispatcher

The dispatcher is responsible to start the process of task execution. Since most of its responsibilities are delegated to additional pluggable services (i.e., remote execution, load distribution and fault-tolerance) this service is devoid of most complexities of the scheduling process.

The current implementation of this service intercepts the execute method from *ITask* and submits the task to an available *IServer*. By default tasks are executed on available resources, in a round-robin strategy. The following pseudo-code illustrates how this functionality can be plugged with AOP into generated *ITask* either by the *FrameworkAdapter* or by the *Parallelisation* service.

Algorithm 2 Dispatcher pseudo-code.

```
around(ITask task) : call( * ITask.execute())
                     && target(task) {  //target for method execution

       IServer server = ... //get server from the resource list
       return server.run(task);
}
```

2.3 Load Distribution

The load distribution service performs a fine tuned mapping among available resources and *ITask*'s. This service is required to manage applications that generate irregular tasks or/and when tasks will run on heterogeneous resources, with, for example, varying processing speeds.

This service changes the default round-robin scheduling strategy to a demand-driven, dispatching more tasks to resources that process tasks faster (either because they receive more lightweight tasks or because they have higher processing capabilities).

The current implementation creates a thread pool to dispatch work for each resource. It ensures that each thread dispatches tasks always to the same resource. Threads are continuously picking work from the work pool, sending the work to the processing resource and waiting for the task to complete.

2.4 Fault-tolerance

This service manages faulty resources by resubmitting tasks for execution when the previously assigned resource fails. This service only addresses resources that fail after a task is assigned. In other cases the resource is simply discarded from the list of available resources (i.e., *IServer* list).

The fault tolerance capability is non-invasively plugged into the dispatcher through a time-out mechanism. When a task is dispatched to a resource (`IServer.run(ITask)`) this service starts a timing mechanism to trigger a time-out event after a predefined time. On a time-out event the target resource may be discarded from the resource list and the task is again resubmitted for execution (by calling `ITask.execute()`). Note that when both load distribution and fault tolerance services are plugged the fault tolerance service intercepts the task dispatching process after the load balancing service (i.e., after the task have been assigned to a free resource), and tasks resubmitted by the fault tolerance service are again assigned to specific resources by the load distribution service.

2.5 Resource List

This module maintains a list of remote resources (i.e., a list of *IServer*). Other modules, such as Load Distribution or Fault-Tolerance or Monitoring, access the list accordingly to their specific needs, those can be getting or updating the list.

Each remote resource must send a message, from time to time announcing that is alive. The content of the message must have all information that makes possible this module to contact the remote server, thus allowing, for example, the submission of tasks. When a message announcing that a remote resource is available arrives, a new server is added to the list, in the other hand, when the submission of a task fails that server can be removed from the list.

2.6 Remote Execution

The remote execution module allows tasks to be executed in distributed processing elements, that is, in different computing nodes, taking advantage of the computing power of a cluster, or even a grid, to execute a set of tasks.

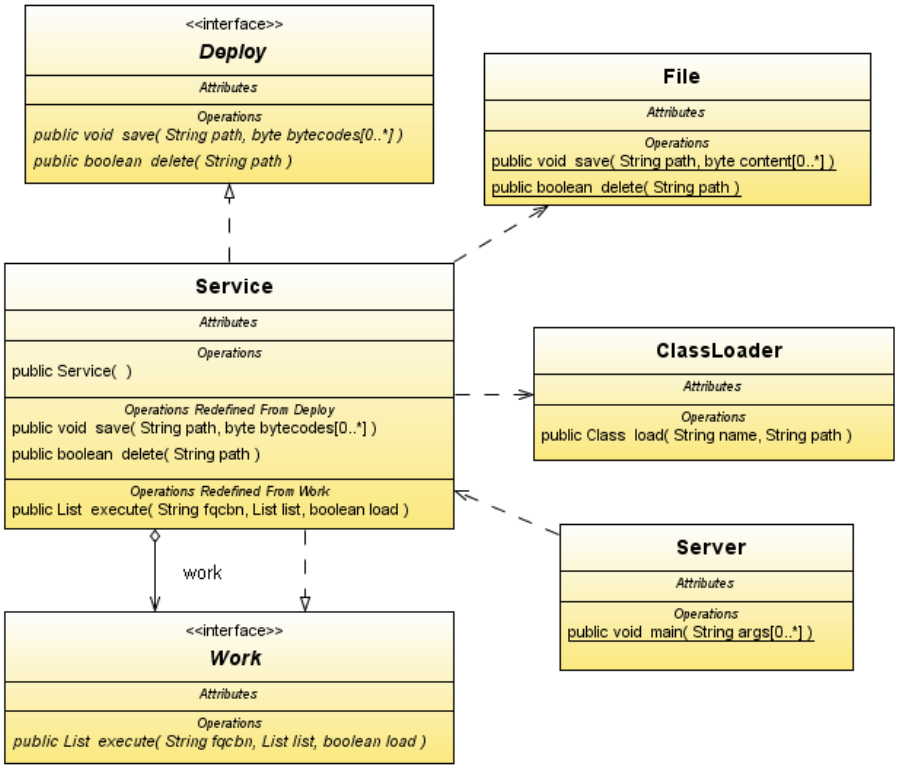

Fig. 4. Class diagram of remote execution module.

Figure 4 presents a class diagram of the remote execution module. Previously to task execution it is necessary to deploy the (specific) code that allows a task to be executed, that is accomplished by transferring the needed bytecode. By means

of a code contract, the bytecode to be transferred has to implement the Work and the Deploy interfaces. Thus, it is possible to execute the deployed code since the implementation of execute method instantiates a Work reference with an instance obtained from the deployed code.

2.7 Monitoring

The monitoring component allows to see the percentage of concluded tasks relative to tasks generated so far, as well as to count the number of task whose execution has failed.

For this purpose, a monitor module intercepts task submission (`ITask.execute(..)`) to gather the number of tasks to execute and monitors calls to the `IServer.run(ITask)` method to find when tasks are dispatched to resources. This second call also allows to detect faulty resources since, in this case task execution does not complete.

3 Evaluation

In this section we illustrate and evaluate the use of the framework to gridify a simple application, the computation of the Mandelbrot set from the Java Grande Forum [15]. The application was non-invasively parallelised with the AspectGrid framework. The Mandelbrot set was divided into disjoint sets, each computed by a different *ITask*.

The non-invasive nature of the AspectGrid framework implies that no overhead is introduced into the original sequential code, as gridification modules can be plugged any time during and after the gridification process. As such, when executing the code on a sequential machine no overhead is observable.

Figure 5 presents execution times (in miliseconds) of the computation when running on a multi-core machine with 4 cores (dual Xeon 2.8GHz) for a varying number of generated parallel tasks. These results where collected by plugging the parallelisation and the load distribution services.

The application benefits from a moderate number of parallel tasks (e.g., few tens) since they generate tasks with varying workloads. Increasing the number of tasks makes it possible to perform a better load distribution up to a point where this is an excess of parallel tasks.

Figure 6 presents execution times (in milliseconds) of the applications when running on a cluster with 16 machines without the load distribution service.

Figure 7 presents a screenshot of the current implementation of the monitoring service. It shows the overall progress of the computation as well as the ratio of faulty tasks. In this case, faulty tasks were artificially injected to test the fault tolerance service.

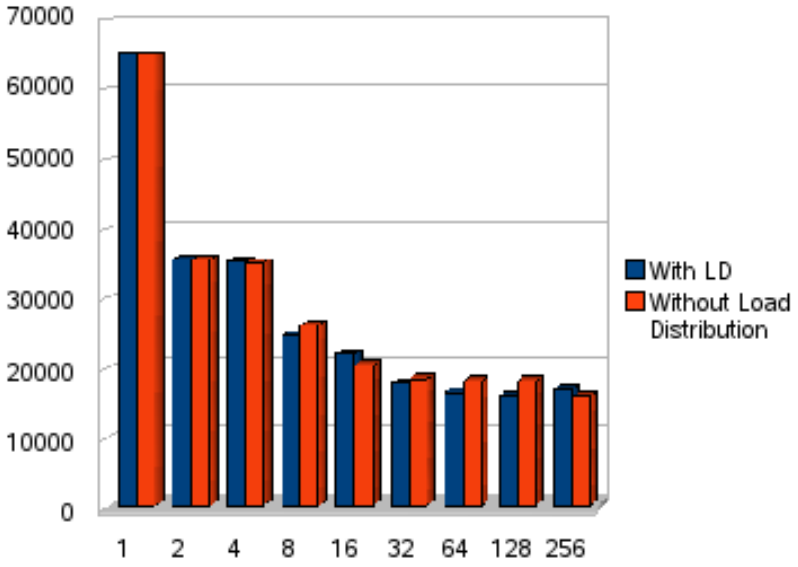

Fig. 5. Execution times on a multi-core platform.

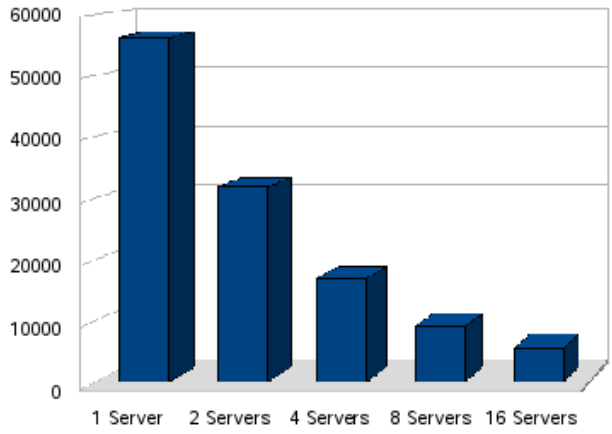

Fig. 6. Execution times for distributed version.

4 Related work

The Java GAT [4] is a grid API that aims to provide a simple interface to multiple grid middleware. Ibis [18], ProActive [5] and HOCs [10] provide middleware to develop parallel applications that can take advantage of grid systems. Gridgain [2] is an open source framework designed specifically to support the development of grid applications. Grid-enabling applications in these approaches require invasive

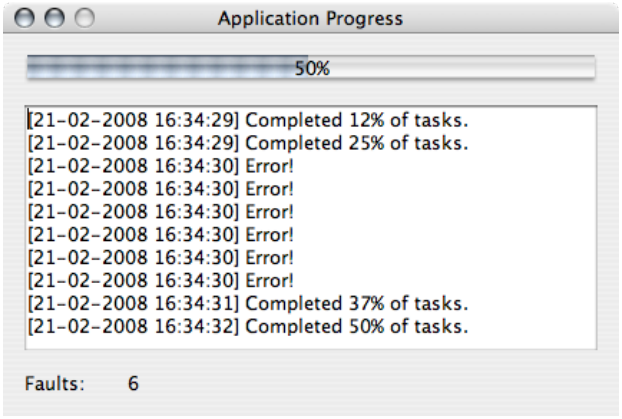

Fig. 7. Monitor window.

and non-reversible source code changes. In these approaches grid-enabled scientific applications become dependent of the Grid middleware.

GEMLCA [6] and GRASG [11] are two frameworks supporting non-invasive gridification of scientific codes. These approaches perform a coarse grain gridification, by deploying scientific codes as grid services. These approaches lack of support for fined-grained decomposition of the application functionality to take advantage of the power of computational grids.

Non-invasive fine-grained gridification has been previously applied to applications that adhere to specific coding conventions. The Pagis system [19] explores the use of reflection techniques to gridify applications structured according to the paradigm of process networks. AOP techniques have been previously applied to abstract the process of remote execution of Java Thread-based applications [13] and to implement the adaptation of a skeleton framework [7] to cluster and grid environments [17]. The AspectGrid framework differs from these previous efforts by supporting non-invasive, fine-grained, gridification of scientific codes, without requiring the source code to adhere to specific coding conventions. In addition, gridification issues are pluggable, supporting the adaptation of the application to specific running conditions, including the execution on a sequential machine, on multi-core systems and on computational Grids.

5 Conclusion

The AspectGrid framework is a pioneer lightweight framework supporting a set of gridification services provided by the use of AOP techniques. It uses a unique non-invasive approach to bind grid services into scientific codes and to couple together several services provided by the framework, resulting in a non-invasive, fine-grained gridification.

Current work includes the integration of the AspectGrid services with a grid middleware (e.g., Glite) and the extension of the current set of services to address

security issues. More long-term work includes addressing portability issues namely by supporting the gridification of non-Java applications.

Acknowledgments

This work was supported by AspectGrid project (Pluggable Grid Aspects for Scientific Applications, GRID/GRI/81880/2006) and by SeARCH (Services & Advanced Computing with HTC/HPC, CONC-REEQ/443/EEI/2005), all funded by Portuguese FCT and European funds (FEDER).

References

1. *AspectGrid homepage*. http://gec.di.uminho.pt/aspectgrid.
2. *Gridgain homepage*. http://www.gridgain.com.
3. *JDL - Job Description Language*. http://glite.web.cern.ch/glite/documentation.
4. G. Allen, K. Davis, T. Goodale, A. Hutanu, H. Kaiser, T. Kielmann, A. Merzky, R. van Nieuwpoort, A. Reinefeld, F. Schintke, et al. The grid application toolkit: toward generic and easy application programming interfaces for the grid. *Proceedings of the IEEE*, 93(3):534–550, 2005.
5. L. Baduel, F. Baude, D. Caromel, A. Contes, F. Huet, M. Morel, and R. Quilici. *Grid Computing: Software Environments and Tools*, chapter Programming, Composing, Deploying, for the Grid. Springer-Verlag, 2005.
6. T. Delaitre, T. Kiss, A. Goyeneche, G. Terstyanszky, S. Winter, and P. Kacsuk. GEMLCA: Running Legacy Code Applications as Grid Services. *Journal of Grid Computing*, 3(1):75–90, 2005.
7. J. Ferreira, J. Sobral, and A. Proenca. JaSkel: a java skeleton-based framework for structured cluster and grid computing. *Proceedings of the Sixth IEEE International Symposium on Cluster Computing and the Grid (CCGRID'06)-Volume 00*, pages 301–304, 2006.
8. R. Filman and D. Friedman. Aspect-Oriented Programming is Quantification and Obliviousness. *Workshop on Advanced Separation of Concerns*, 2000, 2000.
9. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 1995.
10. S. Gorlatch and J. Diinnweber. From Grid Middleware to Grid Applications: Bridging the gap with HOCs. *Future Generation Grids: Proceedings of the Workshop on Future Generation Grids, November 1-5, 2004, Dagstuhl, Germany*, 2005.
11. Q. Ho, T. Hung, W. Jie, H. Chan, E. Sindhu, G. Subramaniam, T. Zang, and X. Li. GRASG-a framework for "gridifying" and running applications on service-oriented grids. *Proceedings of the Sixth IEEE International Symposium on Cluster Computing and the Grid (CCGRID'06)-Volume 00*, pages 305–312, 2006.
12. G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. Loingtier, and J. Irwin. Aspect-Oriented Programming. *ECOOP'97–object-oriented Programming: 11th European Conference, Jyväskylä, Finland, June 9-13, 1997: Proceedings*, 1997.
13. P. Maia, N. Mendonça, V. Furtado, W. Cirne, and K. Saikoski. A process for separation of crosscutting grid concerns. *Proceedings of the 2006 ACM symposium on Applied computing*, pages 1569–1574, 2006.
14. C. Mateos, A. Zunino, and M. Campo. A survey on approaches to gridification. *Software: Practice and Experience*, 38:523–556, 2008.

15. L. Smith, J. Bull, and J. Obdrzalek. A Parallel Java Grande Benchmark Suite. *Proceedings of the 2001 ACM/IEEE conference on Supercomputing (CDROM)*, pages 8–8, 2001.
16. J. Sobral. Pluggable grid services. *Grid Computing, 2007 8th IEEE/ACM International Conference on*, pages 113–120, 2007.
17. J. Sobral and A. Proença. Enabling jaskel skeletons for clusters and computational grids. In *IEEE Cluster*. IEEE Press, 2007.
18. R. van Nieuwpoort, J. Maassen, G. Wrzesinska, R. Hofman, C. Jacobs, T. Kielmann, and H. Bal. Ibis: a flexible and efficient Java-based Grid programming environment. *Concurrency and Computation: Practice & Experience*, 17(7):1079–1107, 2005.
19. D. Webb and A. Wendelborn. The PAGIS Grid Application Environment. *International Conference on Computational Science (Lecture Notes in Computer Science*, 2659.