Pluggable Grid Services

João Luís Sobral

Centro de Ciências e Tecnologias da Computação, Departamento de Informática,
Universidade do Minho, Braga, Portugal

jls@di.uminho.pt

***Abstract*— This paper introduces a new concept of pluggable grid service, that provides seamless access to computational grids, based on aspect-oriented techniques. Pluggable grid services avoid explicit calls to grid services in scientific codes, localize grid-specific concerns into well defined modules and can be (un)plugged from scientific codes. Domain specific code becomes oblivious of grid issues, allowing scientists to concentrate on domain specific issues, and to manage grid issues by composing grid-specific modules. This paper presents a collection of pluggable grid services that illustrates the idea and shows how pluggable grid services can grid-enable a skeleton framework.**

I. Introduction

Computational grids provide unparallel computational power to scientists, making it possible to perform larger and more complex simulations and experiments. However, current grid services require a considerable up-front investment to understand and to use a grid application programming interface. Moreover, to enable applications to run on computational grids, scientists lose control of the overall structure of their code and execution of their code becomes dependent of a grid infrastructure (or a ghost local grid). For instance, when using MPICH-G [1] programmers must parallelize their application, splitting application data among MPI process and to use *send* and *receive* primitives to exchange data among processes. The main control flow is now MPI-based, which poses limitations to understanding and evolving grid-enabled codes. These limitations contribute to a shortage of real grid users, as converting applications to run on computational grids requires multiple and non-reversible changes to the original domain specific code.

Aspect-oriented programming [2] (AOP) was proposed to address crosscutting concerns in systems software. AOP makes it possible to localize into well defined modules concerns that are not effectively modularized with other programming techniques. Concerns modularized with AOP techniques include security [3], persistence [4], distribution [5], concurrency [6], parallelization [7] and profiling [8].

This paper explores the use of AOP techniques to support pluggable grid services (PGS). The idea is three-fold:

(i) **Modularity of grid concerns:** to use AOP techniques to localize grid concerns into well defined modules.

(ii) **Pluggability of grid services:** to provide means to grid-enable scientific applications with less changes to the source code than current approaches.

(iii) **Unpluggability of grid services:** to support the development of grid enabled applications that can run when grid-specific modules are not included in the build.

The goal of pluggable grid services is not to provide a new grid middleware, but to leverage the usage of current grid infra-structure, providing a seamless access to grid resources.

This paper starts by an overview of AOP techniques. Section III presents the concept of pluggable grid service and outlines the current Java-based implementation. Section IV illustrates the use of pluggable grid services by showing how these services were used to transform a Java skeleton framework into a grid-enabled framework. Sections V and VI discuss the approach and related work, respectively. Section VII concludes the paper.

II. Overview of Aspect-Oriented Concepts

AOP was proposed to help to modularize the so-called crosscutting concerns in software systems. Crosscutting concerns are system behaviours that are not effectively modularized with other programming techniques. The implementation of crosscutting concerns leads to code tangling - the code that implements the concern is mixed with other application functionality - and scattering - the code that implements the concern is distributed by several modules.

Suppose, for instance, that the programmer wants to add a logging mechanism to an object oriented application, to print the name of each method executed. Using traditional mechanisms it needs to include a print (or a call to a logging API) on every method implementation, resulting in tangling and scattering of the logging concern. AOP techniques make it possible to attach this behaviour with a single statement like the following (an aspect coded in pseudo-AspectJ [9]):

```
aspect logging {
  around execution( * *.*(..) ) {          // on every method execution
      print( thisJoinPoint().methodName() )   // print method name
      proceed()                              //  execute original method
} }
```

The aspect complier will generate code to call the print statement on every method execution (e.g., by including the print statement on each method implementation). This compilation process in called weaving to emphasize the fact that this behaviour is attached to multiple execution points. Execution points where it is possible to attach additional functionality are called join points. By attaching behaviour to a set of join points a module (i.e., aspect) can change the dynamic behaviour of the base system. Most AOP languages also provide means to change the static structure of applications, by modifying the type hierarchy, for instance by inserting a method into a class or by adding a super-class to an existing class.

Quantification and obliviousness [10] were recently proposed as two main distinguishing features of AOP. Quantification allows an aspect to intercept many, non-local, application execution points and to attach aspect-related behaviour at those points (e.g., in the above example to print the method name). AOP languages usually provide a rich set of options to select a subset of available join points, namely, based on the target object type, method name, type and number of parameters. In the presented example the logging facility could be restrict to methods with no return type (i.e., returning a void type), by using *execution(void *.*(..))*.

Obliviousness is the ability to apply aspects to some base code that was not explicitly prepared for that purpose. In the above example the logging aspect does not require any special cooperation from the original source code writer. This way, the original code can be oblivious of the behaviour introduced by the aspect. A consequence of obliviousness is that the original code can run without the additional behaviour. This work refers to this characteristic as pluggability (a term not coined among the AOP community) to reinforce the idea that the original system does not depend on the specific features implemented by aspects, as this behaviour can be attached and detached from the remainder program behaviour.

III. Pluggable Grid Services

The goal of pluggable grid services it to allow scientists to run scientific applications into their desktops and to transparently access to grid resources, when requested. The idea is to plug accesses to grid services into certain application execution points using pluggable modules, instead of inserting calls to a grid application programming interface. These grid services can, at any time, be unplugged from scientific codes.

Table I illustrates the concept of pluggable grid service. The domain specific code is a ray tracer application (code at the left). The pluggable service (remote execution) redirects the execution of *render* methods to a remote resource. The main point is that the remote execution concern is localized into its own module; it can be plugged into the base scientific code without changing the source code and it can be easily removed from the build (i.e., unplugged). Pluggable services can also be used through code annotations, by introducing the annotation @remoteExec before the *render* method definition. Annotations are an intermediate approach since they can be ignored (i.e., the functionally they attach can be unplugged), however, they require changes to the source code and they lead to non-localized grid concerns.

TABLE I
EXAMPLE OF A PLUGGABLE SERVICE

Scientific code	**Remote execution module**

class RayTracer {
  void render(/*…*/) {
    ...
  }
}
	RemoteExec<RayTracer,”render”>

Pluggable grid services rely on AOP techniques to generate a new type-compatible class from a domain specific class, where the additional functionality is attached into certain execution points (this set of join points is called a pointcut). For instance, a class may become a remote class by changing the class implementation, intercepting object creations and method call statements in the original code to redirect their execution to the remote resource [5].

The collection of pluggable grid services currently includes services for remote object creation and remote method calls, method execution scheduling, granularity control and fail-over execution. Table II summarizes currently supported grid services.

TABLE II
PLUGGABLE GRID SERVICES

Service	**Description**
GridSeparate <Class T, Pointcut P>	Class T becomes grid enabled (can be created on a remote grid resource)
Separate <Class T, Pointcut P>	Class T becomes cluster enabled (can be created on a local cluster node)
Schedule <Class T, Pointcut P>	Schedules method execution in available resources
GrainControl <Class T, Pointcut P>	Task coalescing in calls P of class T
FailOver <Class T, Pointcut P>	Failover execution on methods P of class T

A. Grid Separate

The separate service was inspired in the *separate* keyword from Eiffel [11] that specifies that a class can be placed on a remote resource. The *GridSeparate* service transforms a domain specific class *T* into a grid enabled class. When this service is plugged into a domain specific class, instances of that class are created into remote resources and method calls specified by *pointcut P* (a pattern expression that specifies a set of method calls) are redirected to that remote instance.

Currently two underlying protocols are supported to implement remote objects: Remote Method Invocation based (RMI) and Job Submission based (JS).

The RMI based protocol, transforms a class into a RMI class. It can be used to instantiate remote classes into local area nodes and to communicate with these instances through the standard RMI protocol. It uses a standard *ssh* command to start a daemon on each available local resource. This daemon accepts remote (intra-cluster) requests for object creations and remote method calls.

The JS protocol transforms a class into a stand-alone application that can be deployed on a remote grid resource. This application is instantiated using a grid service (e.g., a Globus GRAM service), receives requests through files that are staged-in and saves its results into files that are staged-out to the client machine.

Both RMI and JS services place an object factory into each remote resource to instantiate domain specific classes. After this factory has been created, object creations in the domain specific code are redirected to the remote resource, by intercepting *new* statements. Method calls to these instances

are redirected to the remote instance; either by means of standard RMI mechanisms or through grid file stage-in and stage-out mechanisms.

Both services require that each class becoming grid enabled should follow the standard rules applied to RMI enabled classes, namely, accesses to static or shared variables are not allowed.

The separate service requires access to a list of available resources to instantiate separate classes and the definition of a policy to select one particular resource from the list. The separate service provides default implementations (e.g., round-robin selection) than can be adapted to an application/middleware specific case, for instance to use an external grid service to find available resources.

The important point about the separate service is that it is a (un)pluggable service and that changes to its implementation do not require changes in the domain specific code.

B. Schedule

The goal of the scheduler service is to select the best resource for each task execution. It acts upon a set of object instances to adapt the load distribution according to the available processing power and the computational requirements of each task.

The *Schedule* service applies a load-balancing mechanism to all instances of class *T* and to method calls specified by pointcut *P*. The load-balancing mechanism selects the instance with the shortest execution time to perform the next method execution. Method calls to a scheduled method are delayed until a class instance is available for execution.

The *Schedule* service can be composed with the *Separate* service, scheduling method executions among remote objects to balance the load. In that case, it may be of interest to employ multiple schedulers in an application. For instance, a scheduling level can perform intra-cluster scheduling whereas another level can schedule method executions among clusters, replacing the traditional rule of a grid meta-scheduler.

Current scheduler implementation is based on Java executors. Whenever a scheduled method call is performed the call is encapsulated into a command pattern [12] and it is placed into a shared work queue. A set of working threads (one for each remote object instance) retrieve tasks from the work queue and send these tasks for remote execution. Whenever a task completes execution, a new task is retrieved from the queue and sent to the remote object instance. Since the workload is distributed on demand there is no need to measure the workload of remote compute resources.

Application-specific schedulers can be implemented by extending the provided scheduler service, for instance, to take advantage of compile-time estimations of execution time.

C. Grain Control

The grain control service has two goals: to decide if a resource should be used for task execution and to combine fine-grained tasks into coarse-grain tasks. The idea is to provide fine-grained tasks (sometimes called over-decomposition [13]) that are coalesced at run-time to match resource availability.

The *GrainControl* service applies a granularity control mechanism to all instances of class *T* and to method calls specified by pointcut *P*. For each remote object instance, this service computes the minimum number of tasks that should be combined into a single request for an efficient resource usage. It is based on a 1 to 10 ratio heuristic (i.e., communication costs should be less than 1/10 of computation costs). The grain control service may decide to not to use a resource due to a too high communication overhead.

The grain control service relies on run-time profiling that collects the time spent on each remote task execution. It is based on a probing phase that gathers the computational power and communication costs for each remote resource. A static function can be supplied to provide an estimate of the method complexity, which can be useful when no profiling information is yet available.

Implementation of the grain control service is based on the composite pattern [12]. Each method call is transformed into a command that is inserted into a composite of tasks that are sent as a single unit for remote execution.

The *GrainControl* service can be combined with both *Separate* and *Schedule* services. It is also possible to combine multiple grain-control services into a single application. For instance, it is possible to have a grain-control service for intra-cluster instances and another service for inter-cluster grain control.

D. Fail Over Service

The fail over service can be used to address faulty resources. It provides a time-out mechanism that may re-execute a task into the same or into another grid resource.

The *FailOver* service applies a fault tolerance mechanism to all instances of class *T* and to method calls specified by pointcut *P*. Whenever a time-out occurs, a mechanism similar to the scheduler selects an available resource for task re-execution. This time-out is computed by using run-time profiling data, but it can also be supplied by the programmer.

The fail over service can be combined with grain-size control to re-execute a set of coalesced tasks.

```
public aspect Separate<Class T, Pointcut P> {

  main() {   // main function. registers server in RMI
      T rt = new T();          // creates a new instance of T
      … // register instance in name server
      … // wait for shutdown
      … // un-register instance in RMI name server
  }

  around execution ( T.new ) {
      … // get a reference to the remote instance;
  }

  around execution (P) {
      … // delegate execution to the remote instance
  }

}
```

Fig. 1 – Outline of the implementation of the *Separate* service

E. Implementation Outline

The current implementation is based on a source code to source code generator that generates AspectJ. Generating AspectJ avoids modifying the original source code and reduces the amount of the generated code when compared with similar Java-based tools.

The *Separate* service relies on AspectJ AOP features to change the type hierarchy and to attach additional functionality to certain execution points. Fig. 1 presents the AspectJ pseudo-code of *Separate<Class T, Pointcut P>* service implementation in Java RMI (for clarity, exception handling code and remote factory code are not included).

The generated code introduces a new *main* method, which creates an instance of class *T* and registers this instance into the RMI name server. This instance will serve incoming RMI requests, until a shutdown order is received. The *around execution(T.new)* statement redirects creations of objects of type *T* to a remote resource (it also creates a local fake object and saves the association among local fakes and remote instances into a local hash map, code not shown in the figure). The *around execution(P)* redirects method calls specified by pointcut *P* to the remote instance. In addition to the generated AspectJ code, the implementation also relies on scripts to perform command line requests to remotely execute java code (e.g. jar files).

The *GridSeparate* service is implemented in a similar manner. However, there are two main differences:

1. Communication with the remote object instance is performed by means of file stage-in and stage-out that contain method execution requests and replies.
2. Object instantiation is performed by submitting a task to a remote grid resource, instead of local creating the executable using the remote secure shell.

The *Schedule* service implementation keeps track of all created instances of the target class (also based on the *T.new* join point but it does not creates new instances). On each method call it selects one target remote instance or it places the request into a work queue.

F. Annotations

Annotations provide an alternative way to plug grid services into specific element types, e.g. methods or classes (see Fig. 2).

```
@ClassAnnotation
public class <class_name> {
        //...
        @MethodAnnotation
        public void method_name() { … }
}
```

Fig. 2 – Annotation elements

The collection of grid services can also be used with annotations (see Table 3). Each implementation (presented in the previous section) supports both the traditional and annotation based style of using grid services. AspectJ enables the use of single implementations for both styles.

TABLE III
PLUGGABLE GRID SERVICES ANNOTATIONS

Service	Annotation	Target type
[Grid]Separate	@[Grid]Separate	Class
Schedule	@Schedule(…)	Method
GrainControl	@GrainControl(…)	Method
FailOver	@FailOver(…)	Method

IV. GRID ENABLED JASKEL APPLICATIONS

This section shows how the proposed pluggable grid services were used to transform a Java-based skeleton framework into a grid enabled framework. The resulting framework allows the transparent execution of skeleton applications on local machines and/or on computational grids.

A. JaSkel Overview

The JaSkel framework [14] is based on a set of abstract Java classes that encapsulate pre-set parallelization strategies. To develop an application the programmer selects the classes that best match the application parallelization structure and fills the framework hooks with domain specific functionality. The framework takes care of all aspects of parallel execution, including starting parallel activities, splitting the work into parallel tasks and joining the results.

Fig. 3 illustrates a simple JaSkel farming application. Lines 01-06 declare a *MyWorker* class that implements the domain specific code. The *compute* method receives a task to process and returns the processed task. Lines 07-16 define a *MyFarm* class that creates several workers and includes the *split* and *join* methods to split the original tasks into independent pieces of work and to join the processed pieces of work. Lines 18-23 create a task and a *MyFarm* instance, start the farm activity and gather the results using the *getResult* method. In this code the number of workers can be provided by the programmer or selected by the framework.

```
01   public class MyWorker 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(Object task) {
10             super(task);     // save task into local structure
11             for(int i=0; i<myNumberOfWorkers; i++)
12                  /* … */ = new MyWorker();
13        }
14        Collection split(Object initialTask) { … }  // split
15        Object join(Collection partialResults) { … } // join
16   }
17
18   // main function
19   Task task = … // task to compute
20   MyFarm farmer = new MyFarm(tasks);
21   farmer.eval();              // starts the farming process
22   Object o = farmer.getResult();    // get results
```

Fig. 3 A sample JaSkel task farm

The *Farm eval* method creates several workers, splits the initial task using the supplied *split* method, sends a piece of work to each worker and joins the processed pieces of work using the supplied *join* method.

Using the built-in *Farm* skeleton it is possible to take advantage of multi-core and/or multiprocessor machines, as it spawns one thread per each worker, performing concurrent *compute* calls.

B. Grid Enabled Skeletons

Pluggable grid services allow the transparent execution of JaSkel applications on cluster and/or grid platforms. The *GridSeparate* service can be applied to the *MyWorker* class to transparently introduce remotely executed workers, using the following statement:

```
GridSeparate<MyWorker,"compute">
```

The previous statement indicates that all instances of *MyWorker* class can be created and executed on a remote grid resource and that method calls to the *compute* method should be redirected to that remote instance. As an alternative, it is possible to specify that instances of the *MyWorker* class are distributed across local nodes, using the *Separate* service.

Selecting the faster resource for a particular task can be addressed using the *Schedule* service. For this purpose the number of provided tasks (computed by the *split* method) should be much larger than the number of workers to attain a good load distribution [15]. The scheduler service implements a demand-driven scheme, where faster resources receive more tasks to process. This service can be applied to this example by the following statement:

```
Schedule<MyWorker,"compute">
```

This statement binds the *Schedule* service to *compute* calls of class *MyWorker*. Calls to this method will be delayed until one remote instance is ready to execute the *compute* call, leading to a demand driven task allocation among available resources.

Selecting the ideal number of resources to use for a particular application can be addressed by the *GrainControl* service. This service uses profiling data to compute the minimum size of each task and it can even decide that a particular resource can not be effectively used by an application. This service can be useful in cases where the available resources are much larger than the computational requirements of the application.

The *FailOver* service can be plugged into the previous application in a similar manner

C. Multi-level Skeletons

This section uses a multi-level skeleton application to illustrate the use of multiple *Separate* and *Schedule* services into a single application. The case study (see Fig. 4) of a multi-level skeleton is a *meta-farm* that creates several farms from Fig. 3. A meta-farm is composed of multiple farms, and each farm has its own workers.

```
01  public class MetaFarm extends Farm {
02     …
03     public MetaFarm(Object task) {
04           super(task);       // save task into local structure
05           for(int i=0; i<myNumberOfFarms; i++)
06                 /* … */ = new MyFarm();
07     }
08     Collection split(Object initialTask) { … }  // meta-farm split
09     Object join(Collection partialResults) { … } // meta-farm join
10  }
11
12  // main function
13  Task task = … // task to compute
14  MetaFarm farmer = new MetaFarm(tasks);
15  farmer.eval();                  // starts the farming process
16  Object o = farmer.getResult();        // get results
17
```

Fig. 4 A JaSkel meta-farm

The idea of a meta-farm is to deploy a farm on each grid resource and to use the scheduler service to perform the load balancing among farms.

Using pluggable services it is possible to deploy each *MyFarm* on a grid cluster and to deploy *MyWorker* instancies into local cluster nodes. This is accomplished by the following statements:

```
GridSeparate<MyFarm,"compute">
Separate<MyWorker,"compute">
```

Multiple scheduler services can be applied in a similar manner using the statements:

```
Schedule<MyFarm,"compute">
Schedule<MyWorker,"compute">
```

One scheduling level (*MyFarm* scheduler) is deployed on each local cluster and a meta-scheduler balances the load among clusters (i.e., among *MyFarm* instancies)

D. Pluggable Grid Services Performance

This section aims to evaluate the performance benefits of pluggable grid services, by comparing the performance of a base JaSkel application and grid enabled versions. Performance results presented in this section were collected using two clusters at the Universidade do Minho campus. Each cluster consists of a 5-nodes, dual processor machines (Xeon 3.2 GHz). Single node tests were performed on a Mac Pro Xeon 5130 (with two dual core processors). Single and multi-cluster results are based on pluggable grid services described in section III.

The PGS approach enables the use of a single code base to run an application on a multi-core machine, on a cluster and on a federation of clusters. Depending on the target platform, the programmer just plugs the required service(s).

Fig. 5 compares the execution times of the base JaSkel code and the same base source code with attached pluggable services (*Separate* services). The application is the Java Grande Forum parallel ray tracer (image size of 4000) [16].

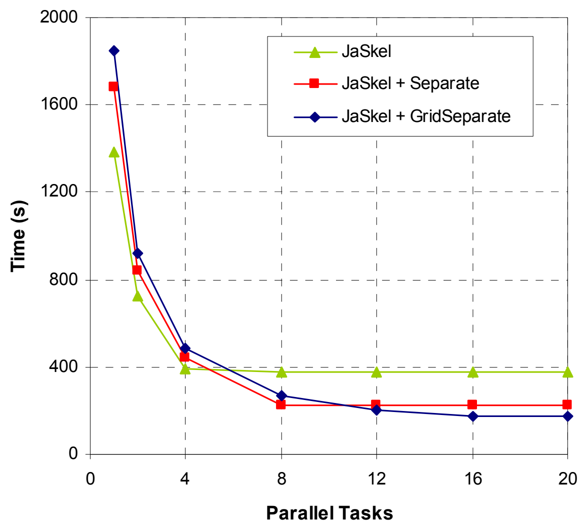

Fig. 5 Separate Grid Services Performance

As it is expected, the base JaSkel application can not take advantage of more than 4 parallel tasks, as it relies on shared memory mechanisms. Using the *Separate* service, JaSkel skeletons can be distributed across the nodes of a local cluster. The *GridSeparate* service distributes tasks among the two clusters, enabling the application to take advantage of more parallel tasks. The *Separate* service imposes an additional overhead when using a small number of nodes, due to the introduction of a communication middleware (e.g., Java RMI). Performance penalty of this service is less than 5%, when compared with the manual introduction of communication primitives into JaSkel skeletons (results not presented here).

The next benchmark evaluates the schedule service. Fig. 6 compares the execution times of the same application (running on two clusters, by means of the *GridSeparate* service), but now there is an imbalance among tasks (the partition of the image to render is made block-wise instead of cyclic, as in the JGF implementation). In this imbalanced version the number of tasks is four times more than the number of remote resources.

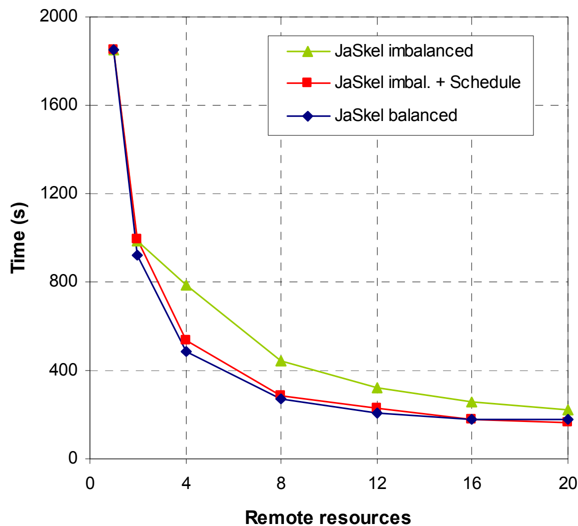

Fig. 6 Scheduler Service Performance

These results show that the scheduler service was able to address the application load imbalances, avoiding execution penalties. This service provides execution times close to the balanced version (5% overhead, on average). Using the schedule service the original application could be relived from the code to provide equally sized tasks. Moreover, the scheduler service can also address the use of heterogeneous remote resources.

The last test evaluates the effectiveness of the grain size control service. Fig. 7. compares execution times of the ray tracer of a smaller image size (1000x1000), when no grain control is performed, when the grain-size is manually tuned and when the Grain-control service is used.

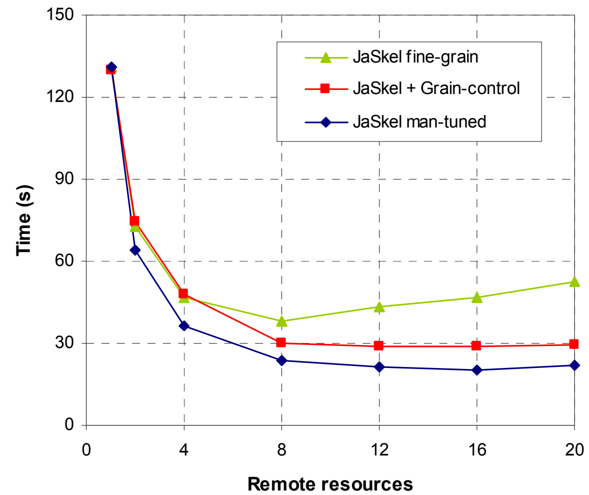

Fig. 7 Grain-control Service Performance

When no grain control is performed (JaSkel fine-grain in the figure) there is an overhead of using a large number of remote resources. This overhead is due to fine-grained tasks that introduce an overhead when they are remotely executed. The Grain-control service can remove this overhead and, more importantly, it can detect that this application can not effectively take advantage of more than 12 remote compute resources, transparently avoiding the use of more resources than required.

V. Discussion

Enabling scientific applications to run on grid platforms with pluggable grid services requires an adequate set of join points (i.e., execution points) in the original code to plug these grid services. Moreover, applications must be amenable to run on grid platforms (i.e., to provide a set of loosely coupled parallel activities, whose execution can be transparently redirected to remote compute resources). If these two requirements are not meet the application must be refactored to expose the required join points and a loosely coupled set of parallel activities. The goal of PGS is not to transform a sequential application into a parallel equivalent, but to help to enable scientific codes to run on computational Grids. Support for these refactorings is delegated to other tools. JaSkel plays an important rule in this path as it provides a software infrastructure to help to perform these refactorings.

Applications built upon the JaSkel farm skeleton already satisfy the above requirements ([7] presents an alternative to JaSkel, using AOP techniques to modularize parallelization strategies). It should be stressed that PGS do not depend on JaSkel or on other tools if the original code is already amenable to run on computational grids.

The current implementation of PGS relies on a library of aspects built on Java and AspectJ. Java greatly simplifies the process of deploying grid applications and PGS since the application and required libraries can be packed into an autonomous jar file and executed on any grid resource that has a Java Virtual Machine (JVM). AspectJ is the most stable and well-know AOP compiler and produces JVM compliant bytecodes. Although it would be possible to fully implement PGS in Java, this approach would replicate weaving tasks currently performed by the AspectJ compiler. The AOP community also provides support for other languages. As an example, AspectC++ [17] is the most well known AOP tool for C/C++. Conceptually, PGS can also be implemented with AOP tools that support other languages.

Supporting PGS for multiple target languages rises a new issue of how the develop language independent PGS than can be weaved into multiple target languages. Ultimately, it would be necessary to build a library of PGS and an aspect weaver for each target language. This issue falls into the more recent field of language independent AOP [18]. These issues are being addressed in the AspectGrid project [19], where PGS will be used to enable Java, C++ and Fortran codes to run on computational grids.

Object-oriented programming fully leverages current AOP techniques. Procedure-based languages, such as C, do not have such strong modularity capabilities; as a consequence they do not provide a so rich set of join points. There are some results showing that legacy systems can also benefit from AOP techniques [20]; however the modularity support of the base language has an impact on the AOP ability to modularize crosscutting concerns. In the specific case of PGS, the set of available join points defines the points in execution where PGS can be attached. A poor set of join points in the target language can make it more difficult to modularize and to support pluggable grid concerns. For instance, in Java/AspectJ an object creation is a join point, but there is no similar join point in C.

Current AOP technology, including the AspectJ compiler, requires access to the application source code. It would be technically possible to perform weaving only at binary level (i.e., without access to application source code) if binaries include the information required for aspect weaving. Languages based on a virtual machine make this task easier since binary files include more symbolic information than binaries generated from languages like C++.

Grid systems are moving towards service oriented architectures. PGS can contribute to this move in two ways: by transparently enabling applications to consume grid services and by helping to expose certain functionality as a grid service. First, PGS help to decouple the client from the service provider (actually the client is oblivious of the provided grid service), making it easier to transparently delegate a specific functionality to an external service provider. For instance, the fail-over PGS could be transparently delegated to an external provider. Second, PGS can help to transparently deploy certain application functionally as a grid service. For instance, the *GridSeparate* PGS deploys an instance of a Java class into a remote resource. This approach can be extended to deploy a Java class as a web service on a remote container.

VI. RELATED WORK

The Globus toolkit [21] is a standard for grid computing. It includes services for authentication and authorization, resource discovery, allocation and monitoring and file management. The Java commodity grid kit (Java CoG) [22] provides an application programming interface to grid services. It provides a set of Java interface components, mapping commonly used grid services from Globus and also includes utility components and services to improve Globus toolkit functionality. GMarte [23] is a high level Java API that offers an object-oriented, user friendly, interface to the Globus. It allows programmers to run applications on grid environments without detailing the procedure of job execution. The Grid Application Toolkit (GAT) [24] aims to provide a simple, clear interface to many different grid infrastructures, by providing a set of adaptors that connect GAT to grid services, supporting multiple providers (e.g., Globus, Unicore). These approaches require an up-front application redesign to use the provided grid application programmer interface.

MPICH-G2 [1] enables the transparent execution of MPI programs into computational grids. Higher-Order Components for grids [25] is skeleton framework specifically designed to support the execution of skeleton-based applications on grid systems. The use of AOP techniques to transparently execute Java thread-based applications on grids is addressed in [26]. PGS differ from these approaches, since PGS address grid issues by means of modular and (un)pluggable services.

Approaches based on XML-based descriptors, commonly used in large-scale client/server applications, were proposed to deploy applications on grid environments [27]. In this approach, applications are developed for a set of virtual nodes and XML-based descriptors are used to map virtual nodes into physical resources, without source code changes. GridGain [28] is open source framework to support the development of grid applications. It provides the *@Gridify* annotation to specify execution into a remote grid resource. Both approaches are closely related to our approach, as they provide a seamless access to grid resources. Pluggable grid services can be more easily composed and extended to deploy efficient grid enabled applications. Both XML-based descriptors and annotations must resort to other techniques when the service must be configured with application specific behaviour. These approaches do not aim to attain modular and (un)pluggable grid services. PGS goes a step further by completely relying modular and pluggable grid services to manage grid concerns.

VII. CONCLUSION

This article presented a new approach to make grid enabled scientific codes, based on pluggable grid services. The approach relies on aspect-oriented techniques to localize grid concerns into (un)pluggable modules.

Pluggable grid services promote a softer transition to grids as they can be attached to current scientific applications without requiring source code modifications. Pluggable grid services can be flexibly composed to develop applications that can take advantage of the hierarchical nature of computational grids. Moreover, pluggable grid services can be unplugged from scientific applications making it easier to evolve grid enabled codes.

Experimentation with the JaSkel framework showed how these goals were achieved: using the proposed approach JaSkel applications transparently become grid-enabled, by plugging modules that localize grid execution issues. This migration did not required changes to the framework code and original JaSkel applications can still run when grid related modules are unplugged.

Current work includes the development of a complete set of language independent pluggable grid services, namely to address data grids, utility computing and security issues. This set of services will be more tightly connected to current grid technology, for instance, the scheduler implementation can use current grid schedulers, instead of implementing its own scheduling policy.

ACKNOWLEDGMENTS

The author would like to acknowledge to the anonymous reviewers and to Carlos Cunha for their comments that helped to clarify the explanation of the Pluggable Grid Service concept.

This work was supported by PPC-VM project (Portable Parallel Computing Based on Virtual Machines, POSI/CHS/47158/2002), 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] N. Karonis, B. Toonen, and I. Foster, "MPICH-G2: A Grid-Enabled Implementation of the Message Passing Interface", *Journal of Parallel and Distributed Computing,* vol. 63, no. 5, May 2003.

[2] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. Loingtier, J. Irwin, "Aspect-Oriented Programming" *European Conference on Object-Oriented Programming (ECOOP),* 1997.

[3] R. Hao, L. Bölöni, K. Jun, D. Marinescu, "An aspect-oriented approach to distributed object security", *Fourth IEEE Symposium on Computers and Communications*, 1999.

[4] A. Rashid, R. Chitchyan, "Persistence as an aspect", *2nd International Conference on Aspect-oriented Software Development (AOSD)*, 2003.

[5] S. Soares, L. Loureiro, P. Borba, "Implementing Distribution and Persistence Aspects With AspectJ", *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'02)*, Nov. 2002.

[6] C. Cunha, J. Sobral, M. Monteiro, "Reusable Aspect-Oriented Implementation of Concurrency Patterns and Mechanisms", *Fifth International Conference on Aspect Oriented Software Development (AOSD'06)*, March 2006.

[7] J. Sobral, "Incrementally developing parallel applications with AspectJ", *20th IEEE International Parallel & Distributed Processing Symposium (IPDPS'06)*, Greece, Rhodes, April 2006.

[8] D. Pearce, M. Webster, R.t Berry and P. Kelly, "Profiling with AspectJ", *Software: Practice and Experience*, vol. 37, no. 7, June 2007.

[9] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, W. Griswold, "An Overview of AspectJ", *European Conference on Object-Oriented Programming (ECOOP)*, June 2001.

[10] R. Filman, D. Friedman, "Aspect-Oriented Programming is Quantification and Obliviousness", In *R. Filman et al., (Eds.), Aspect-Oriented Software Development*, Addison-Wesley, 2005.

[11] B. Meyer, *Object-Oriented Software Construction*, Prentice Hall 1997

[12] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Reading, MA: Addison-Wesley, 1995.

[13] R. Fernandes, K. Pingali and P. Stodghill, "Mobile MPI Programs in Computational Grids", *ACM Symposium on Principles and Practices of Parallel Programming (PPoPP)*, 2006.

[14] J. Fernando, J. Sobral, A. Proenca, "JaSkel: A Java Skeleton-Based Framework for Structured Cluster and Grid Computing", *6th IEEE International Symposium on Cluster Computing and the Grid* (CCGrid'2006), Singapore, May 2006.

[15] I. Foster, *Designing and Building Parallel Programs*, Addison-Wesley, 1995.

[16] J. Smith, J. Bull, J. Obdrzálek, "A Parallel Java Grande Benchmark Suite", *Supercomputing Conference (SC 2001)*, Denver, Nov. 2001.

[17] www.aspectc.org

[18] D. Lafferty, V. Cahill, "Language-Independent Aspect-Oriented Programming", *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, Anaheim, CA, October 2003.

[19] gec.di.uminho.pt/aspectgrid

[20] R. Lämmel, K. Schutter, "What does aspect oriented programming mean to Cobol?", *International Conference on Aspect-Oriented Software Development (AOSD)*, Chicago, IL, March 2005

[21] www.globus.org

[22] G. Laszewski. I. Foster, J. Gawor, P. Lane, "A Java commodity grid Kit", *Concurrency and Computation: practice and experience,* vol. 13, no. 8-9, 2001.

[23] J. Alonso, V. Hernández, G. Moltó, **"**GMarte: Grid middleware to abstract remote task execution", *Concurrency and Computation: Practice and Experience*, vol. 18, no. 15, December 2006.

[24] G. Allen, K. Davis, T. Goodale, A. Hutanu, H. Kaiser, T. Kielmann, A. Merzky, R. van Nieuwpoort, A. Reinefeld, F. Schintke, T. Schott, E. Seidel, B. Ullmer, "The grid application toolkit: toward generic and easy application programming interfaces for the grid", *Proceedings of the IEEE*, vol. 93, no. 3, March 2005.

[25] S. Gorlatch and J. Dunnweber, "From Grid Middleware to Grid Applications: Bridging the Gap with HOCs", *Future Generation Grids*, Springer, 2006.

[26] P. Maia, N. Mendonça, V. Furtado, W. Cirne, K. Saikoski, "A Process for Separation of Crosscutting Grid Concerns", *ACM Symposium of Applied Computing* (SAC'06), April 2006.

[27] F. Baude, D. Caromel, F. Huel, L. Mestre, J. Vayssière, "Interactive and Descriptor-based Deployment of Object-Oriented Grid Applications", *11th IEEE International Symposium on High Performance Distributed Computing*, Edinburgh, Scotland, July, 2002.

[28] www.gridgain.com