On the Evaluation of Gridification Effort and Runtime Aspects of JGRIM Applications. (C. Mateos, A. Zunino, M. Campo). Future Generation Computer Systems: The International Journal of Grid Computing: Theory, Methods and Applications. Elsevier Science. ISSN: 0167-739X. Ed.: P. Sloot. Vol. 26, Num. 6, pp 797-819. 2010. DOI: 10.1016/j.future.2010.02.014

On the Evaluation of Gridification Effort and Runtime Aspects of JGRIM Applications

Cristian Mateos*, Alejandro Zunino, Marcelo Campo

ISISTAN Research Institute. UNICEN University. Campus Universitario, Tandil (B7001BBO), Buenos Aires, Argentina. Tel.: +54 (2293) 439682 ext. 35. Fax.: +54 (2293) 439683
Also Consejo Nacional de Investigaciones Científicas y Técnicas (CONICET)

Abstract

Nowadays, Grid Computing has been widely recognized as the next big thing in distributed software development. Grid technologies allow developers to implement massively distributed applications with enormous demands for resources such as processing power, data and network bandwidth. Despite the important benefits Grid Computing offers, contemporary approaches for Grid-enabling applications still force developers to invest much effort into manually providing code to discover and access Grid resources and services. Moreover, the outcome of this task is usually software that is polluted with Grid-aware code, thus maintainability suffers. In a previous article we presented JGRIM, a novel approach to easily gridify Java applications. In this paper, we report a detailed evaluation of JGRIM that was conducted by comparing it with Ibis and ProActive, two platforms for Grid development. Specifically, we used these three platforms for gridifying the k-Nearest Neighbor algorithm and an application for restoring panoramic images. The results show that JGRIM simplifies gridification without resigning performance for these applications.

Key words: Grid Computing, Gridification, Grid services, JGRIM, Dependency Injection

1. Introduction

The term "Grid" refers to a widely distributed computing environment whose main purpose is to meet the increasing demands of advanced science and engineering [1, 2]. Within a Grid, hardware and software resources from distributed sites are virtualized to transparently provide applications with vast amounts of resources. Just like an electrical power grid, a computational Grid offers a powerful yet easy-to-use computing infrastructure to which applications can be plugged and efficiently executed without much effort from the user [3]. Unfortunately, given the extremely heterogeneous, complex nature inherent to Grids, writing and adapting applications to execute on a Grid is certainly a very difficult task. This raises the challenge of providing appropriate techniques to *gridify* applications, that is, semi-automatic and ideally automatic methods for easily transforming conventional applications to applications that are capable of benefiting from Grid resources.

In this light, a number of tools for simplifying Grid application development have been proposed. Basically, the goal of these technologies is to unburden developers of the necessity to know the many particularities to contact individual Grid services (e.g. protocols and endpoints), capture common patterns of service composition (e.g. secure data transfer), and offer convenient programming abstractions (e.g. master-worker templates). Roughly, these programming tools can be grouped into toolkits and frameworks. On one hand, the idea behind programming toolkits is to provide high-level programming APIs that abstract away the details of the services provided by existing Grid platforms. Examples of these tools include GSBL [4], Java CoG Kit [5], MyCoG.NET [6], GAT [7] and SAGA [8]. On the other hand, Grid programming frameworks capture common Grid-dependent code and design in an application-independent manner (e.g. application

*Corresponding author.
Email address: cmateos2006@gmail.com (Cristian Mateos)

On the Evaluation of Gridification Effort and Runtime Aspects of JGRIM Applications. (C. Mateos, A. Zunino, M. Campo). Future Generation Computer Systems: The International Journal of Grid Computing: Theory, Methods and Applications. Elsevier Science. ISSN: 0167-739X. Ed.: P. Sloot. Vol. 26, Num. 6, pp 797-819. 2010. DOI: 10.1016/j.future.2010.02.014

templates, service composition patterns, etc.) and provide *slots* where programmers can put application specific functionality to build a complete Grid application. Examples of such frameworks include MW [9], AMWAT [10] and JaSkel [11].

A remarkable feature of the above technologies is that gridification is accomplished through a *one--step* process, that is, there is not a clear separation between the tasks of writing the pure functional code of an application and adding Grid concerns to it [12]. Developers Grid-enable applications as they write code by keeping in mind specific API calls or framework constructs. Hence, technologies promoting one-step gridification assume developers have a solid knowledge on the programming tool being used. Alternatively, there are some efforts promoting a *two-step* methodology to gridification (see [12] for a comprehensive discussion of them), which are mostly aimed at supporting developers having little or even no background on Grid technologies. Basically, the ultimate goal of approaches falling in this category is to allow developers to incorporate Grid behavior to an application *after* the logic of the application has been implemented. Consequently, projects in this line of research are mainly intended to provide gridification methods rather than Grid programming facilities.

In a previous paper, we proposed JGRIM [13], a two-step gridification method for gridifying Java applications. Specifically, we described the features of JGRIM and showed its practical advantages through preliminary experiments based on source code metrics. In this paper, we report a thorough evaluation of the approach by comparing gridification effort as well as execution performance and network resource utilization with respect to related approaches. To this end, two existing applications were gridified and deployed on an Internet-based Grid. In order to assess gridification effort, we introduce a novel metric called GE (Gridification Effort) that takes into account the amount of redesign, reimplementation and deployment effort necessary to port an ordinary application to a Grid, which is independent of the gridification tool. All in all, the experiments will contribute to have a better understanding of the benefits and potentials of our approach for porting applications to a Grid, and executing the resulting Grid-enabled code.

The rest of the paper is organized as follows. The next section discusses the most relevant related work. Then, Section 3 takes a deeper look at the concepts and notions underpinning JGRIM. The section also illustrates through code examples the facilities offered by JGRIM for gridification. Later, Section 4 presents a detailed evaluation of the approach, which represents the main goal of the paper. Lastly, Section 5 concludes the paper.

2. Related work

Several approaches for gridifying conventional software can be found in the literature. Next, we briefly describe the approaches that are more relevant to our work.

Ibis [14] is a Grid platform for implementing Java-based applications. Ibis is designed as an uniform, extensible and portable communication library on top of which a variety of popular programming models such as MPI [15] and RMI [16] have been implemented. Another interesting programming model of Ibis is Satin [17, 18], which allows developers to parallelize CPU-bound, divide and conquer applications. Satin is aimed at exploiting CPU resources, but does not provide support for taking advantage of other types of Grid resources such as services, data repositories, applications, etc. Finally, Ibis does not offer support for Web Service technologies such as WSDL [19] and UDDI [20]. Indeed, Web Services and generally speaking Service Oriented Architectures (SOAs) play a very important role in Grid Computing because they address the problem of heterogeneous systems integration Stockinger [21]. These technologies thus supply the basis for more interoperable Grids and underlie several of the current Grid initiatives Munawar et al. [22].

ProActive [23] is a Java platform for parallel distributed computing. Applications are composed of mobile entities called *active objects* (AO). AOs serve method calls originated from other AOs and regular Java objects based on the *wait-by-necessity* mechanism, which asynchronously handles individual calls, and transparently blocks requesters upon the first attempt to read the result of an unfinished call. ProActive also provides *technical services* [24], a flexible support that allow developers to address non-functional concerns (e.g. load balancing and fault tolerance) by plugging certain configuration to applications at deployment time. A drawback of ProActive is that AO creation, lookup and mobility are in charge of the programmer. Therefore, the code for managing parallelism and AO migration is mixed with the application

On the Evaluation of Gridification Effort and Runtime Aspects of JGRIM Applications. (C. Mateos, A. Zunino, M. Campo). Future Generation Computer Systems: The International Journal of Grid Computing: Theory, Methods and Applications. Elsevier Science. ISSN: 0167-739X. Ed.: P. Sloot. Vol. 26, Num. 6, pp 797-819. 2010. DOI: 10.1016/j.future.2010.02.014

logic. Besides, ProActive provides limited/no support for Web Service invocation/discovery. Similarly, JavaSymphony [25] deals semi-automatically with migration and parallelism, allowing programmers to explicitly control such features as needed. However, JavaSymphony also suffers from the problems of mixing these non-functional concerns with functional ones, rendering gridification difficult. Like Ibis, JavaSymphony offers limited support for using common Grid protocols and technologies.

XCAT [26] supports distributed execution of component-based applications. An XCAT application is a stateful functional Grid service comprising several components. XCAT runs on top of existing Grid platforms (e.g. Globus [27]), linking individual components to concrete platform-level services. Besides, application components can also represent legacy binary programs. XCAT provides an API that allows developers to build complex applications by assembling service components and legacy components. Though this task can be carried out with little coding effort, developers still have to programmatically manage component creation and linking at the application level. Furthermore, opportunities for application tuning largely depend on the facilities the underlying Grid platform being used offers, as XCAT does not provide support for fine tuning components at the application level.

Despite greatly simplifying the process of adapting the code of an application for Grid enabling it, these approaches still require a significant amount of coding effort from the developer. As an alternative, there are tools that follow what we might call a "gridify as is" philosophy. These approaches treat an input application as a black box by taking either its source or executable code, along with some user--provided configuration (e.g. input arguments, CPU/memory requirements, etc.), and wrap this code with components that isolate the details of the underlying Grid [28, 29, 30, 31, 32]. In this way, the requirement of source code modification when gridifying applications is eliminated. However, output applications are coarse grained, monolithic Grid applications whose structure cannot be altered to make better use of Grid resources. For example, most of these approaches do not prescribe mechanisms for parallelizing or distributing individual application components.

Approaches relying on an API-inspired approach to gridification unavoidably require modifications to the source code of the original application Mateos et al. [13]. Therefore, in many cases, the resulting application code is harder to maintain. However, developers have more control of the internal structure of their applications. On the other hand, the approaches based on wrapping techniques simplify gridification, but in general prevent the usage of tuning mechanisms. This represents a tradeoff between ease of gridification versus flexibility to configure the runtime aspects of gridified applications [12]. Precisely, JGRIM targets this tradeoff by avoiding excessive code modification or provisioning when porting applications to a Grid, nonetheless providing means to tune Grid applications. JGRIM preserves the integrity of the application logic by encouraging developers to concentrate on coding the functionality of applications, and then non--intrusively adding Grid concerns to them. Its core API only have to be explicitly used when performing application tuning. Finally, because of its component based roots, using JGRIM does not differ too much from using popular programming models for Java such as JavaBeans[1] or EJB[2].

3. The JGRIM approach

JGRIM Mateos et al. [13] is an approach for easily porting applications to service-oriented Grids. JGRIM simplifies the construction of Grid applications by allowing users to focus first on the development of the application logic without worrying about common Grid-related concerns such as resource discovery, service invocation and execution management. Essentially, JGRIM lets applications to discover and use the services offered by a Grid without the need to explicitly provide code for either finding or accessing these services from within the application logic.

At the heart of JGRIM is a semi-automatic *gridification process* that developers have to follow to gridify applications. This process accepts as input ordinary component-based applications, where components do not share any state and are described through well-defined interfaces, and transforms these applications to codes which are furnished with specialized components called *metaservices*. Metaservices allows developers to take advantage of existing Grid services, but minimizing the knowledge required to carry out

[1] JavaBeans `http://java.sun.com/javase/technologies/desktop/javabeans/docs/spec.html`
[2] Enterprise JavaBeans `http://java.sun.com/products/ejb`

On the Evaluation of Gridification Effort and Runtime Aspects of JGRIM Applications. (C. Mateos, A. Zunino, M. Campo). Future Generation Computer Systems: The International Journal of Grid Computing: Theory, Methods and Applications. Elsevier Science. ISSN: 0167-739X. Ed.: P. Sloot. Vol. 26, Num. 6, pp 797-819. 2010. DOI: 10.1016/j.future.2010.02.014

gridification. From now on, we will refer to such transformed applications as *gridified* or *Grid-enabled* applications. Figure 1 depicts an overview of JGRIM. As illustrated, JGRIM adds an intermediate Meta-

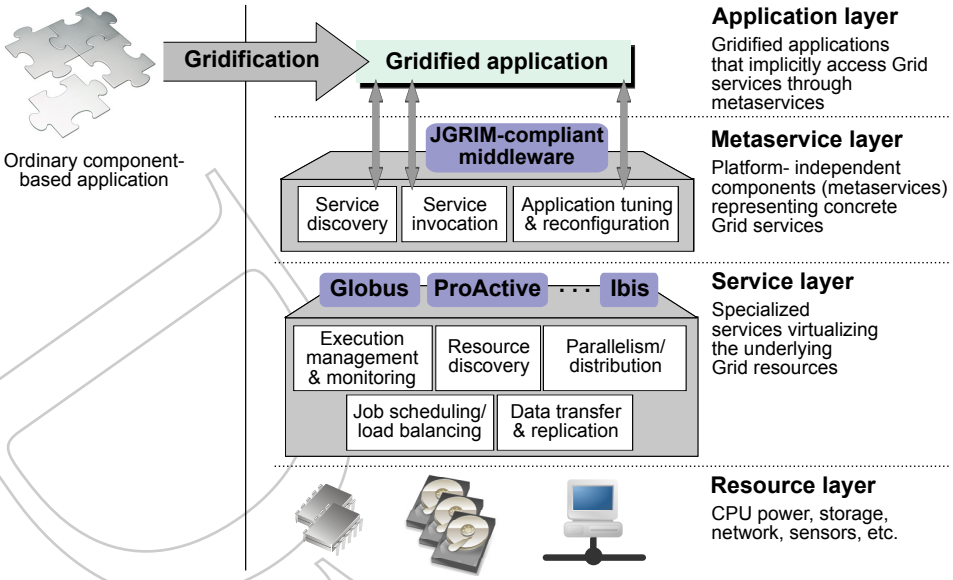


Figure 1: An overview of JGRIM

service layer that enables gridified applications to seamlessly use existing Grid services. In middleware terms, the execution of gridified applications is subject to a software/hardware stack comprising:

- *Resource layer*: Represents the physical infrastructure of a Grid, given by resources such as computing nodes, networking facilities and storage systems, along with the necessary low-level protocols to interact with them (e.g. TCP/IP).
- *Service layer*: Provides, by means of Grid platforms such as Globus, Condor [33], Ibis [14] and ProActive [23], a catalog of services (e.g. job scheduling, load balancing, parallelism) to make effective and efficient use of Grid resources. These platforms offer sophisticated Grid functionalities that are accessible to applications through specific protocols and APIs.
- *Metaservice layer*: Is composed of a number of *metaservices* that act as a glue between gridified applications and the Grid. Metaservices isolate Grid-enabled applications from technology-related details for accessing the Service layer (i.e. the aforementioned protocols and APIs), and can be understood as representatives of related concrete services, that is, those providing similar Grid functionality. JGRIM provides metaservices for:
  - *Service Discovery*: This metaservice can talk, for example, to a UDDI registry or the MDS-2 Globus system [34] to find a list of required Grid services, and then present the results to Grid applications in a common format. This metaservice hides all concrete lookup Grid services behind a generic, technology-neutral lookup(serviceInterface) primitive.
  - *Service Invocation*: Isolates applications from the technologies involved in invoking Grid services or, in other words, provides a generic call(serviceDescriptor) primitive. Typically, the information to interact with an individual service (i.e. datatype formats, binding parameters, etc.) are specified in a service descriptor such as a WSDL document.
  - *Application Tuning*: This metaservice is associated to certain application elements to improve performance. For example, all invocations performed on a mission critical computation can be submitted to a Globus environment, increasing fault tolerance. Similarly, divide and conquer or embarrassingly parallel operations can be executed concurrently to achieve better performance

On the Evaluation of Gridification Effort and Runtime Aspects of JGRIM Applications. (C. Mateos, A. Zunino, M. Campo). Future Generation Computer Systems: The International Journal of Grid Computing: Theory, Methods and Applications. Elsevier Science. ISSN: 0167-739X. Ed.: P. Sloot. Vol. 26, Num. 6, pp 797-819. 2010. DOI: 10.1016/j.future.2010.02.014

and scalability. Then, the purpose of this metaservice is to make the interaction between the individual components of gridified applications more efficient and robust by leveraging existing Grid services for job execution, parallelism and mobility. The metaservice also materializes *policies* Mateos et al. [35], that is, non-intrusive mechanisms by which developers can customize the way an application behaves in a Grid.

- *Application layer*: Contains Grid-enabled applications that implicitly exploit Grid services. During gridification, JGRIM alters some of the original application components and their interactions by using metaservices, so that at runtime some internal operation requests originated by applications at this layer are handled by metaservices instead.

The next subsections give a down-to-earth explanation of the concepts underpinning JGRIM and its associated gridification process for Grid-enabling ordinary applications.

3.1. Associating ordinary applications with Grid services

Unlike many of the existing approaches to gridification, in which users have to explicitly alter the application code to use Grid services [12], the aim of JGRIM is to non-intrusively incorporate Grid services into the logic of ordinary applications. The assumption that drives this idea is that it is possible to associate Grid services to the various *dependencies* of individual software components. A component C_1 has a dependency to another component C_2 if C_1 explicitly uses any of the operations of C_2[3]. JGRIM is based upon associating Grid services to dependencies via metaservices. In this way, interacting components can indirectly benefit from Grid services without changing their internal implementation.

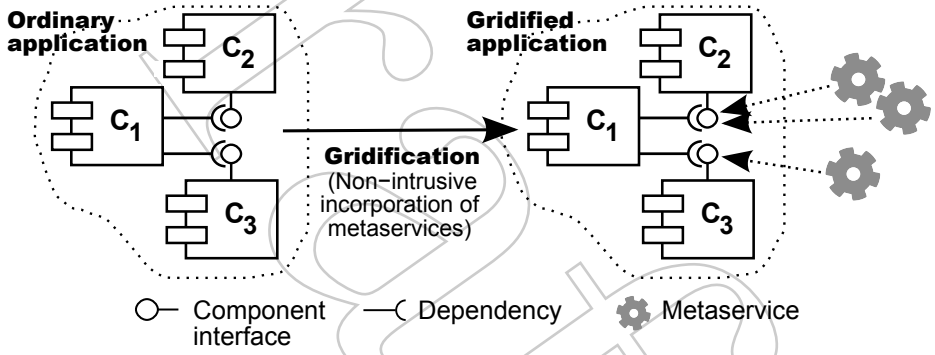


Figure 2: Component dependencies and metaservices

Figure 2 summarizes the concepts exposed so far. Components of an ordinary application are designed to interact with each other by means of their interfaces, thus establishing dependencies. Upon gridification, individual dependencies are associated with one or more metaservices, which control how a dependency is managed within a Grid setting. At the implementation level, dependencies and metaservices are associated through Dependency Injection (DI) [36], a technique that allows software components to be externally and non-invasively supplied with concrete implementations for their dependencies. For further details on the usage of DI for metaservice injection, and the mechanisms by which metaservices are bridged to Grid services see [13].

An interesting implication of dependencies is that, besides denoting links between application components, they can be employed to refer to external functionality upon which one or more components depend but for which an application does not provide an implementation. In fact, Grids often offer services not only for managing hardware resources, but also for providing functional capabilities (e.g. numerical algorithms, search engines, visualization software, etc.) that play the role of third-party building blocks for creating new applications. Then, JGRIM distinguishes between two types of Grid services:

[3]Note that the concept of dependency is not exclusive to component-based software, but it is also present in any other programming paradigm.

On the Evaluation of Gridification Effort and Runtime Aspects of JGRIM Applications. (C. Mateos, A. Zunino, M. Campo). Future Generation Computer Systems: The International Journal of Grid Computing: Theory, Methods and Applications. Elsevier Science. ISSN: 0167-739X. Ed.: P. Sloot. Vol. 26, Num. 6, pp 797-819. 2010. DOI: 10.1016/j.future.2010.02.014

- *Functional Grid service (FGS)*: Is a callable service interface that mediates the access to one or more software and data resources, such as a wrapper to a legacy program or a facade to various data repositories. FGSs expose their functionality through clear interfaces (i.e. with well-defined operations signatures) and, for interoperability and standardization purposes, are often implemented with Web Services technologies [37]. Moreover, after gridification, any conventional application may become itself a functional service, which other applications can discover and use. JGRIM exploits FGSs through the Service Discovery and the Service Invocation metaservices.

  Nowadays, Grid standards like OGSA [38] and WSRF [39] heavily rely on the notion of explicitly interfaced service for materializing platform-level services. This has motivated the evolution of Grid platforms from their pre-Web Service state to new versions based on Web Services [37], being Globus and Condor examples of such evolved platforms. These platforms expose their APIs as Web Services. Two representative examples are the Globus WS GRAM job submission service[4] and more recently the gLite CREAM job management service Aiftimiei et al. [40]. However, this kind of services should not be confused with FGSs. The main difference is that the former are located at the Basic layer, whereas the latter are located at the Application layer.

- *Non-functional Grid service (NFGS)*: NFGSs differ from FGSs in that the latter are used as *functional* building blocks for applications. NFGSs are further classified into interfaced platform-level services, and those services representing abstract Grid concerns for which a clear and standard interface to their capabilities cannot be specified. An example of interfaced NFGS is Globus WS GRAM. Examples falling into the latter group are parallelism, distribution, mobility, load balancing, fault tolerance, security, among others. NFGSs are located at the Basic layer. An individual NFGS may have many materializations. For instance, Globus and Condor provide job submission services. Similarly, ProActive and Ibis implement parallelism. JGRIM exploits NFGSs through the Application Tuning metaservice.

Table 1: FGSs versus NFGSs

	FGSs	**NFGSs**
Located at	Application layer	Service layer
Purpose	To provide functional building blocks	To address non-functional concerns
Explicitly interfaced?	Always	Not always
Exploited through	Service Discovery and Service Invocation metaservices	Application Tuning metaservice

Table 1 summarizes the two discussed types of Grid services. Note that exploiting functional Grid services may imply the use of non-functional Grid services as well (e.g. using a *secure* mechanism to contact an FGS), but not the other way around. Both FGSs and NFGSs are proxied and represented by JGRIM through metaservices, which are injected into the dependencies between ordinary application components. During the process of gridification, the developer is responsible for selecting which of these dependencies and components are enhanced with Grid capabilities. The next subsection explains this process.

3.2. Gridification Process

In addition to metaservice provisioning, a fundamental aspect of JGRIM is its *gridification process*, this is, the set of tasks developers have to follow to adapt their applications to run on a Grid. The JGRIM gridification process is illustrated in Figure 3, and consists of the following steps:

[4]WS GRAM `http://www.globus.org/toolkit/docs/4.0/execution/wsgram`

On the Evaluation of Gridification Effort and Runtime Aspects of JGRIM Applications. (C. Mateos, A. Zunino, M. Campo). Future Generation Computer Systems: The International Journal of Grid Computing: Theory, Methods and Applications. Elsevier Science. ISSN: 0167-739X. Ed.: P. Sloot. Vol. 26, Num. 6, pp 797-819. 2010. DOI: 10.1016/j.future.2010.02.014

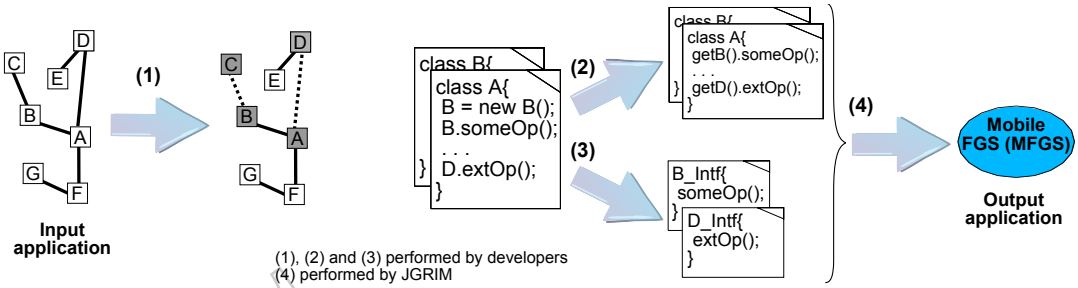


Figure 3: JGRIM: Gridification process

1. *Hot-spot identification*: A developer identify the particular dependencies and components that will benefit from the Grid, which are basically the *hot-spots* for gridification within the application into which metaservices are injected. In the figure, hot-spots have been sketched with dashed lines. For example, the implementation of component C may be outsourced to an FGS. Therefore, the dependency from B to C may be equipped with runtime service discovery and invocation. The dependency A-D may be enhanced, for instance, with fault tolerance.
   Components like C, whose implementation is outsourced to an FGS, are known as *external*. Conversely, components for which an implementation is supplied by the application (e.g. A, B, D) are called *internal*. A dependency involving two internal components is an internal dependency, whereas an external dependency originates when an internal component accesses an external one. JGRIM also defines another type of dependency called *self*, which models the case when a component uses its own methods to implement operations.
2. *Coding conventions*: This step involves the modification of an application code to ensure that the application components interact between each other through get/set accessors using the JavaBeans style. Any reference to a component Comp within the source code must be done by calling a fictitious method getComp(), instead of accessing it directly as Comp.operation(). For example, if an application reads data from a file component, it should be accessed as getFile().read(). Then, JGRIM takes the modified component code and introduces some minor modifications into it so as to add the necessary support for DI.
3. *Component interface definition*: In this step, the developer specifies the interfaces of the internal and external application components to separate what a component does from its implementation. For the internal interfaces, this is a common practice in Java, thus most of the time this task is not necessary. For the external interfaces, it involves defining the method signatures of the third-party functional services used by the application.
4. *Assembly and deployment*: JGRIM combines the outputs of (2) and (3) and deploys the newly gridified application on a Grid. The resulting application is a mobile functional Grid service (MFGS), which migrates its execution based on environmental conditions [35] such as CPU/memory availability, network latency and bandwidth, etc. Anatomically, an MFGS is composed of a non-mobile and a mobile part. The non-mobile part is given by a WSDL document describing the interface of the MFGS (automatically built from the public methods of the original application) and a proxy. This proxy resides on a specific host of the Grid and is in charge of acting as a bridge between the WSDL and the mobile part of the MFGS. The mobile part is Grid-enabled code with migratory capabilities that accesses Grid services through JGRIM metaservices.

The process assumes input applications as being designed under a component-based paradigm. Though this paradigm is very popular among Java developers, the assumption does not hold for any kind of Java application, for example legacy codes or applications having a monolithic structure in terms of object-oriented design. Nevertheless, the problem of componentizing object-oriented applications has been already addressed Lee et al. [41], Kim and Chang [42], Li and Tahvildari [43]. We are investigating a similar approach to supply the above process with an extra transformation step to ensure, prior to the step number (1), that input applications are component-based.

On the Evaluation of Gridification Effort and Runtime Aspects of JGRIM Applications. (C. Mateos, A. Zunino, M. Campo). Future Generation Computer Systems: The International Journal of Grid Computing: Theory, Methods and Applications. Elsevier Science. ISSN: 0167-739X. Ed.: P. Sloot. Vol. 26, Num. 6, pp 797-819. 2010. DOI: 10.1016/j.future.2010.02.014

3.3. An example: Panoramic image restoration

This section describes the gridification of an application for restoring panoramic images. Target images are located at a remote repository. Anatomically, the application is structured as a master-worker application. The master operates by downloading, via FTP, an image from the repository. Then, it splits the image into smaller images, assigns each subimage to a worker for restoration, and then joins the results. Finally, the master encodes the restored image into a specific bitmap format (e.g. JPEG, PNG) that is passed as a parameter to the restoration process. In the non-gridified application, the input image is split into two halves. This enables the application to take advantage of dual core and biprocessor CPUs.

Suppose we have already implemented some of the components of the application but without keeping in mind Grid concerns. Roughly, these components include an ImageRestorer declaring two operations (restoreImage and restoreSubimage), an FTPClient implementing operations for obtaining file metadata (size, permissions, etc.) and transferring files, and a number of helper classes. The idea is then to take the source code, along with some user-provided configuration (mainly for deployment purposes), and generate the corresponding gridified application. Since the original application does not provide a component for image encoding, we will outsource an implementation from the Grid.

The ImageRestorer component materializes the above master that coordinates the whole restoration process. This component uses the operations of FTPClient to download files. Enhancement of individual halves of the input image is separately handled by two concurrent threads (workers) that execute the restoreSubimage method. After the two subimages have been processed, ImageRestorer contacts an external encoder component to generate the final image in the desired format:

```
public class ImageRestorer {
  private FTPClient ftpClient;

  public ImageRestorer(){
    ftpClient = new FTPClient();
  }
  public byte[] restoreImage(String imageURI, String format) {
    ftpClient.transferFile(imageURI, "/tmp/tmpImage");
    byte[] imageData = loadImage("/tmp/tmpImage");
    byte[][] halves = split(imageData);
    // Create and start worker threads
    WorkerThread worker0 = new WorkerThread(this, halves[0]);
    WorkerThread worker1 = new WorkerThread(this, halves[1]);
    worker0.start(); worker1.start();
    // Wait until child threads are finished
    worker0.join(); worker1.join();
    byte[] result = combine(worker0.getResult(), worker1.getResult());
    // Interaction with an external component
    return encoder.encode(result, format);
  }
  public byte[] restoreSubimage(byte[] imageData) {...}
}

public class FTPClient {
  public FileMetadata getMetadata(String fileURI) {...}
  public void transferFile(String fileURI, String localDir) {...}
}

public class WorkerThread extends Thread {
  private ImageRestorer restorer = null;
  private byte[] half = null;
  private byte[] result = null;

  public WorkerThread(ImageRestorer restorer, byte[] half) {
    this.restorer = restorer;
    this.half = half;
  }
  public void run() {
    result = restorer.restoreSubimage(half);
  }
  public byte[] getResult() {
    return this.result;
```

On the Evaluation of Gridification Effort and Runtime Aspects of JGRIM Applications. (C. Mateos, A. Zunino, M. Campo). Future Generation Computer Systems: The International Journal of Grid Computing: Theory, Methods and Applications. Elsevier Science. ISSN: 0167-739X. Ed.: P. Sloot. Vol. 26, Num. 6, pp 797-819. 2010. DOI: 10.1016/j.future.2010.02.014

```
  }
}
```

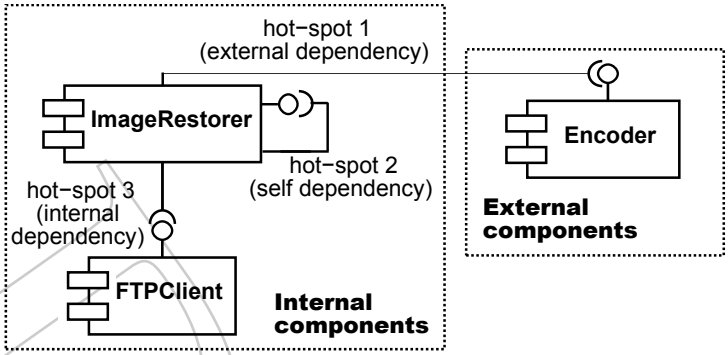


Figure 4: The image restoration application: components and dependencies

Figure 4 illustrates the components of the application and the relationships between them. For the sake of clarity, we will not follow the gridification process in the exact order presented in Section 3.2. Instead, we will take one hot-spot at a time from the figure and incrementally carry out the subsequent gridification activities, that is, steps (2) and (3) of the gridification process.

3.3.1. Hot-spot 1: The ImageRestorer-encoder dependency

The first hot-spot is the ImageRestorer-encoder dependency. We have to provide the expected interface at the client-side for the encoding service, and then alter the code of ImageRestorer so that all invocations to operations of the encoder component are performed through a call to a fictitious getEncoder() method. JGRIM will then automatically inject metaservices capable of dynamically discovering an FGS adhering to that interface. In the example, the expected operations for encoder have been defined in ImageEncoderIF. Processing this information with JGRIM results in:

```
1  public class ImageRestorer extends jgrim.core.MFGS {
2    ...
3    ImageEncoderIF encoder = null;
4
5    public ImageEncoderIF getEnconder() {
6      return this.enconder;
7    }
8    public void setEncoder(ImageEncoderIF encoder) {
9      this.encoder = encoder;
10   }
11   public byte[] restoreImage(String imageURI, String format) {
12     ...
13     // Implicit interaction with an FGS
14     return getEncoder().encode(result, format);
15   }
16 }
17
18 public interface ImageEncoderIF {
19   public byte[] encode(byte[] imageData, String format);
20 }
```

Gridified applications extend the MFGS class, which implements basic primitives for performing application mobility and exporting application methods as Web Services. Besides, note that JGRIM also added a new instance variable (encoder) and proper getter/setters methods for accessing this variable (lines 3-10). These instructions allows JGRIM to transparently incorporate service discovery and invocation capabilities to ImageRestorer by means of DI. Basically, these metaservices are implemented by classes that provide runtime inspection of UDDI registries and invocation of Web Services, which are associated to the application through an automatically generated file:

On the Evaluation of Gridification Effort and Runtime Aspects of JGRIM Applications. (C. Mateos, A. Zunino, M. Campo). Future Generation Computer Systems: The International Journal of Grid Computing: Theory, Methods and Applications. Elsevier Science. ISSN: 0167-739X. Ed.: P. Sloot. Vol. 26, Num. 6, pp 797-819. 2010. DOI: 10.1016/j.future.2010.02.014

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN 2.5//EN"
  "http://www.springframework.org/dtd/spring-beans-2.5.dtd">
<beans>
  <bean id="mainComponent" class="ImageRestorer">
    <property name="encoder">
      <ref local="encoderMetaService"/>
    </property>
  </bean>
  <bean id="encoderMetaService" class="jgrim.core.JGRIMServiceProxy">
    <property name="expectedInterface" value="ImageEncoderIF"/>
  </bean>
</beans>
```

The configuration file links the aforementioned classes together forming a fully operative application. The two benefits of this approach are that components are heavily decoupled, since binding to external Grid services is performed at runtime, and the source code is free from instructions for finding and invoking these services. In addition, it is very easy to replace the reference to the encoding service for testing purposes by simply replacing the encoderMetaService component in the configuration file with, for example, a mock object. Currently, JGRIM is based on Spring [36], a popular DI framework in which all of the wiring is performed by means of XML files.

3.3.2. Hot-spot 2: The ImageRestorer-ImageRestorer dependency

The restoreImage operation of ImageRestorer has been implemented by issuing two different asynchronous invocations to restoreSubimage, and then combining the results. These calls are inherently independent between each other, hence they have been implemented to execute concurrently by using threads. Grids offer many alternatives to threads to handle the execution of parallel computations. These alternatives come in the form of sophisticated services that, besides parallelism, include desirable features such as load balancing, data distribution, fault tolerance, and so on. In JGRIM, these services can be exploited through self dependencies.

Self dependencies originate when a method of a component calls other methods of the same component. In this way, applying the steps (2) and (3) of the JGRIM gridification process to a self dependency results in: (a) using get accessors when invoking any embarrassingly parallel method like restoreSubimage, and (b) defining an interface including all those methods that are subject to concurrent execution. In our example, we will name this interface SubImageRestorerIF. Then, the gridified source code is:

```
1  public class ImageRestorer extends jgrim.core.MFGS {
2    ...
3    SubImageRestorerIF self = null;
4
5    public SubImageRestorerIF getSelf() {
6      return this.self;
7    }
8    public void setSelf(SubImageRestorerIF self) {
9      this.self = self;
10   }
11   public byte[] restoreImage(String imageURI, String format) {
12     ...
13     byte[][] halves = split(imageData);
14     byte[] result0 = getSelf().restoreSubimage(halves[0]);
15     byte[] result1 = getSelf().restoreSubimage(halves[1]);
16     // Wait until computations are finished
17     byte[] result = combine(result0, result1);
18     ...
19   }
20 }
21
22 public interface SubImageRestorerIF {
23   public byte[] restoreSubimage(byte[] imageData);
24 }
```

To Grid-enable the self dependency, the programmer must replace the asynchronous invocations to the restoreSubimage operation within restoreImage by sequential calls to the same operation through the

On the Evaluation of Gridification Effort and Runtime Aspects of JGRIM Applications. (C. Mateos, A. Zunino, M. Campo). Future Generation Computer Systems: The International Journal of Grid Computing: Theory, Methods and Applications. Elsevier Science. ISSN: 0167-739X. Ed.: P. Sloot. Vol. 26, Num. 6, pp 797-819. 2010. DOI: 10.1016/j.future.2010.02.014

self component (lines 14-15). Similar to the case of the image encoding service, JGRIM automatically adds a metaservice definition to the configuration file previously shown, and appends the code to support DI for this component (lines 3-10). The only extra programming convention needed for the mechanism to work is that the results of the parallel computations must be placed on two different local Java variables (lines 14-15). Further references to any of these results (e.g. line 17) will transparently block the execution of restoreImage until they are computed by the metaservice, which dynamically intercepts both calls and executes them concurrently. Behind scenes, JGRIM further modifies the source code implementing the container method (i.e. restoreImage) to add the necessary instructions to support this synchronization mechanism by relying on the concurrency API of the Java 2 Platform.

The advantages of the gridified code are twofold. One one hand, it is free from threading code, thus it is more clean and legible. Even more important, execution of spawned methods can be handled by using any of the existing Grid job submission services, which are implemented to exploit distributed processors. At present, JGRIM utilizes the parallelization and scheduling services of the Satin subsystem of Ibis. In addition, the materialization of metaservices for using the execution services of ProActive and Condor is underway. This integration is in principle viable from a technical point of view, since ProActive is also implemented in Java, and an interface to Condor clusters already exist for this language [44].

3.3.3. Hot-spot 3: The ImageRestorer-FTPClient dependency

The last hot-spot for gridification in the restoration application is the dependency between the ImageRestorer and FTPClient components. Once more, direct usage of FTPClient is disallowed, thus we have to replace any access to this object within ImageRestorer by calls to the corresponding getter method (e.g. getDownloader()). Furthermore, we have to isolate the implementation of FTPClient behind a generic interface, for example FileDownloader. Note that this practice helps in creating more flexible component designs. For instance, another concrete component for file downloading (e.g. based on GridFTP [45] instead of plain old FTP) can be easily configured, provided this new component adheres to FileDownloader, this is, the service interface required by the application.

As mentioned in past paragraphs, with JGRIM, gridified applications are deployed on a Grid in the form of mobile functional services. Broadly, mobility can bring significant benefits in terms of decreased latency and bandwidth usage when applications are moved to locally interact with remote data. Particularly, our example application could use mobility for accessing the remote image repository. For example, an interesting performance improvement may be to move the application to the repository location in those cases in which the size of the target image file exceeds a certain threshold. In JGRIM, this kind of performance tweaks are introduced by means of *policies*. A policy is a component that mediates between the two components involved in an internal or external dependency. In this sense, mobile behavior can be added to the application by associating the following policy to the analyzed hot-spot:

```
1  import jgrim.core.MFGS;
2
3  public class MovePolicy extends jgrim.policy.PolicyAdapter {
4    public void executeBefore() {
5      // Obtains the file to download from the execution context of
6      // getDownloader().transferFile(imageURI, "/tmp/tmpImage")
7      Object[] args = getExecContext().getMethodInvocation().getArguments();
8      String fileURI = (String)args[0];
9      FileDownloader downloader = (FileDownloader)getExecContext().getTargetComponent();
10     long fileSize = downloader.getMetadata(fileURI).getSize();
11     if (fileSize > 1048576) {
12       MFGS app = (MFGS)getExecContext().getSourceComponent();
13       // Move ImageRestorer to the repository location
14       app.moveTo(parseServerLocation(fileURI));
15     }
16   }
17 }
```

Upon each interaction between ImageRestorer and FileDownloader (i.e. the source and target components of the hot-spot, respectively), MovePolicy is evaluated by JGRIM. The code within executeBefore is run just before the execution of an individual operation defined in FileDownloader takes place. Analogously, developers can specify an executeAfter method. The reified information about the execution

On the Evaluation of Gridification Effort and Runtime Aspects of JGRIM Applications. (C. Mateos, A. Zunino, M. Campo). Future Generation Computer Systems: The International Journal of Grid Computing: Theory, Methods and Applications. Elsevier Science. ISSN: 0167-739X. Ed.: P. Sloot. Vol. 26, Num. 6, pp 797-819. 2010. DOI: 10.1016/j.future.2010.02.014

of operations (source and target components, argument values, etc.) is made accessible to programmers through the getExecContext method from the policy framework (lines 7, 9 and 12). Concretely, the policy moves the application to the site hosting the data (line 13) if the size of the image file about to be downloaded is greater than 1 Mb. If that is the case, the "downloading" process will be started locally at that host.

Policies are configured to act upon invocations on specific operations of a target component. In particular, we want MovePolicy to be activated only when the transferFile operation is called. Consequently, the configuration file for the application results in:

```
1  <?xml version="1.0" encoding="UTF-8" ?>
2  <!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN 2.5//EN"
3    "http://www.springframework.org/dtd/spring-beans-2.5.dtd">
4  <beans>
5    <bean id="mainComponent" class="ImageRestorer">
6      ...
7      <property name="downloader">
8        <ref local="policyMetaservice"/>
9      </property>
10   </bean>
11   ...
12   <bean id="policyMetaservice"
13       class="org.springframework.aop.framework.ProxyFactoryBean">
14     <property name="proxyInterfaces" value="FileDownloader"/>
15     <property name="interceptorNames">
16       <list>
17         <value>policyExecutor</value>
18         <value>downloader</value>
19       </list>
20     </property>
21   </bean>
22   <bean id="policyExecutor" class="jgrim.core.PolicyExecutor">
23     <property name="policy">
24       <ref local="movePolicy"/>
25     </property>
26     <property name="activateOn">
27       <list>
28         <value>transferFile</value>
29       </list>
30     </property>
31   </bean>
32   <bean id="movePolicy" class="MovePolicy"/>
33   <bean id="downloader" class="FTPClient"/>
34 </beans>
```

JGRIM injects into ImageRestorer a policyMetaservice component (line 8), which intercepts the calls to methods defined in FileDownloader (line 14) and delegates their execution first to the policy (line 17) and then to the target component (line 18, the FTP client). Methods whose execution must be intercepted by the metaservice are listed in the activateOn property (line 26).

Besides being useful for non-invasively introducing mobility to applications, policies are also employed to customize the way an application interacts with external FGSs. For example, let us suppose a JGRIM application is deployed on a Grid of several sites each hosting a replica of an FGS-wrapped database, which is accessed by the Grid-enabled application. Additionally, let us assume that bandwidth between sites could drastically vary along time. Under these conditions, bandwidth may significantly affect application response time. Particularly, accessing a database replica through a busy network link might decrease performance. Through a policy, we can control which candidate FGS is chosen for serving each application request and, for example, select the service instance that offers the best transfer capabilities.

The rest of the paper contains a detailed evaluation of JGRIM so as to assess and report the practical benefits of the gridification process exposed so far.

On the Evaluation of Gridification Effort and Runtime Aspects of JGRIM Applications. (C. Mateos, A. Zunino, M. Campo). Future Generation Computer Systems: The International Journal of Grid Computing: Theory, Methods and Applications. Elsevier Science. ISSN: 0167-739X. Ed.: P. Sloot. Vol. 26, Num. 6, pp 797-819. 2010. DOI: 10.1016/j.future.2010.02.014

4. Experimental evaluation

This section presents experiments that were carried out in order to provide empirical evidence about the practical soundness of JGRIM. In [13], we conducted a preliminary comparison between JGRIM and similar approaches to gridification based on classic source code metrics. Here, we report a more detailed evaluation of JGRIM by quantifying gridification effort, and measuring execution time and network usage.

In short, ProActive, Ibis and JGRIM were employed to gridify existing implementations of two different applications, namely the k-NN algorithm [46] and the application for enhancement of panoramic pictures presented previously. After gridification, code metrics on the Grid-enabled applications were taken to quantitatively assess how hard is to port these applications to our Grid setting with either of the three alternatives. In addition, experiments were conducted to test various runtime aspects of the resulting Grid applications.

In the next subsection, we describe the characteristics of our experimental Grid setting. Then, subsections 4.2 and 4.3 analyze the gridification of the k-NN algorithm and the image restoration application, respectively.

4.1. The Grid setting

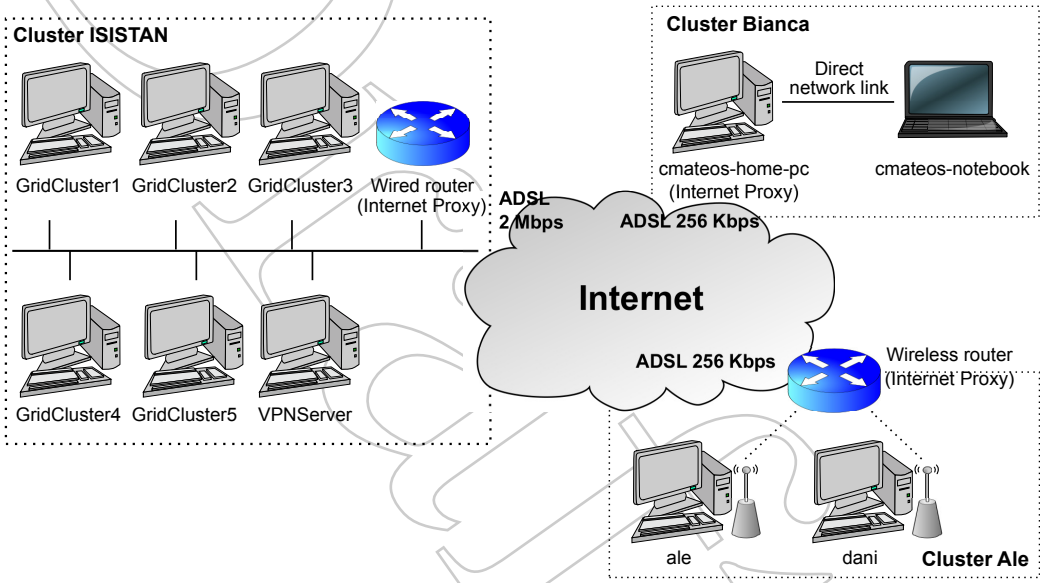


Figure 5: Grid used for the experiments

Figure 5 depicts the anatomy of the Grid that was used to run the experiments. Specifically, three clusters named ISISTAN, Bianca and Ale were linked through a virtual private network by using OpenVPN [47], a software for creating point-to-point encrypted tunnels between Internet-connected computers. The machines of the ISISTAN cluster are part of a larger, public network and share the 2 MB ADSL Internet link. In contrast, Bianca and Ale clusters are local private networks with exclusive access to their associated links.

An OpenVPN server was installed on the "VPNServer" machine (ISISTAN cluster) thus establishing a simple star configuration for the VPN. Preliminary network benchmarks showed that, on average, communication between nodes of the Bianca or the Ale cluster and any machine of the ISISTAN cluster experienced a latency in the range of 60-90 milliseconds, whereas communication between Bianca and Ale clusters was subject to a latency of 100-150 milliseconds. All tests described further in this paper were performed during night time (approximately from 11 P.M. to 8 A.M.), that is, when Internet traffic is lower and network latency is less variable. Furthermore, all tests were launched from the "ale" machine.

On the Evaluation of Gridification Effort and Runtime Aspects of JGRIM Applications. (C. Mateos, A. Zunino, M. Campo). Future Generation Computer Systems: The International Journal of Grid Computing: Theory, Methods and Applications. Elsevier Science. ISSN: 0167-739X. Ed.: P. Sloot. Vol. 26, Num. 6, pp 797-819. 2010. DOI: 10.1016/j.future.2010.02.014

Table 2: CPU and memory specifications of the Grid machines

Machine name	CPU model	CPU frequency	Memory (MB)
GridCluster1	Pentium III (Coppermine)	852 Mhz.	256
GridCluster2	Pentium III (Coppermine)	852 Mhz.	256
GridCluster3	Pentium III (Coppermine)	852 Mhz.	384
GridCluster4	Pentium III (Coppermine)	852 Mhz.	384
GridCluster5	Pentium III (Coppermine)	798 Mhz.	256
VPNServer	Intel Pentium 4	2.80 Ghz.	512
cmateos-home-pc	AMD Athlon XP 2200+	1.75 Ghz.	256
cmateos-notebook	Intel Core2 T5600	1.83 Ghz. (per core)	1.024
ale	AMD Athlon 64 X2 Dual Core 3.600+	2 Ghz. (per core)	1.024
dani	AMD Sempron	1.9 Ghz.	512

Table 2 details the hardware specifications (CPU[5] and memory) of the nodes of our Grid setting. Machines were equipped with Ubuntu Linux 2.6.20 and the Sun JDK 1.5.0. Finally, to keep the system clocks accurately synchronized across the Grid, NTP[6] was used. NTP is a protocol for synchronizing the clocks of computer systems over packet-switched networks that is designed to resist the effects of variable latency. Both CPU power and memory availability varied not only among different clusters, but also among machines of the same cluster. Similarly, communication bandwidth and latency across clusters were quite different. In fact, Grids are characterized by being an arrangement of (usually very) heterogeneous machines connected through network infrastructures with different capabilities. In this sense, we used a testbed with these characteristics so as to perform the experiments on a realistic Grid setting.

4.2. The k-NN algorithm

The k-nearest neighbor algorithm (k-NN for short) is a supervised learning technique. k-NN classifies a new object (or instance) by assigning to it the most common class label among its k nearest neighbors in a multidimensional feature space (or dataset). In other words, k-NN finds the k objects (or training instances) which are closest to a given input instance. The algorithm is CPU-intensive, hence it is a suitable application to be deployed on a Grid.

Suppose we have a dataset containing data about paper tissues, where each training instance is represented by two features (acid durability and strength) and a class label that indicates whether a particular tissue is good or not. Table 3 shows four training samples of this hypothetical dataset. Now, if a factory produces a new paper tissue that pass a laboratory test with acid durability = 3 and strength = 7, k-NN can establish the quality of the paper based on the new sample and the information stored in the dataset.

The k parameter of k-NN is a positive integer, typically small. The best choice of k depends upon the data. Usually, larger values of k reduce the effect of noise on the classification, but make boundaries between classes less distinct. A good k can be selected by various heuristic techniques (e.g. cross-validation). However, to experiment with time-consuming computations, performance tests used a large, fixed value for k.

A pseudo code of k-NN is shown in Algorithm 1. The original version of the application was implemented as a single Java class accessing a file-based dataset through a Dataset class. The application provided two operations classifyInstance and classifyInstances for classifying one instance and a list of

[5]Dual core features were disabled due to limitations in the current implementation of JGRIM

[6]The Network Time Protocol project `http://www.ntp.org`

On the Evaluation of Gridification Effort and Runtime Aspects of JGRIM Applications. (C. Mateos, A. Zunino, M. Campo). Future Generation Computer Systems: The International Journal of Grid Computing: Theory, Methods and Applications. Elsevier Science. ISSN: 0167-739X. Ed.: P. Sloot. Vol. 26, Num. 6, pp 797-819. 2010. DOI: 10.1016/j.future.2010.02.014

Table 3: A sample dataset with four training instances

Acid durability (seconds)	**Strength (kg/square meter)**	**Class (quality)**
7	7	Bad
7	4	Bad
3	4	Good
1	4	Good

Algorithm 1 The k-nearest neighbor algorithm

```
procedure CLASSIFY(instance, k)                    ▷ Returns the class label associated to an instance
    double[] attrs ← GETATTRIBUTES(instance)
    Vector neighbors ← INITIALIZENEIGHBORSLIST(k)
    for all trainingInstance ∈ Dataset do
        double[] trainingAttrs ← GETATTRIBUTES(trainingInstance)
        String trainingClassLabel ← GETCLASSLABEL(trainingInstance)
        double distance ← EUCLIDEANDISTANCE(instance, trainingInstance)
        SORTEDINSERTION(neighbors,distance,trainingClassLabel)    ▷ neighbors is kept sorted
(smaller distances first)
        if LISTSIZE(neighbors) > k then
            REMOVELAST(neighbors)
        end if
    end for
    return MOSTFREQUENTLABEL(neighbors)
end procedure
```

instances, respectively. The latter was implemented as a loop that iterates the input list and calls classifyInstance. On the other hand, Dataset included a method for reading a block of training instances and a method for obtaining the number of instances of the dataset. During gridification, Dataset was replaced with an FGS, and parallelism and policies were introduced into the application.

The dataset file was wrapped with a Web Service exposing analogous operations to those implemented by the Dataset class, and a replica of this service (along with the associated data) was deployed on each cluster of the above Grid. The dataset contained 10.000 records, each described by 20 attributes with randomly-generated numerical values, and a numerical class label representing three predefined categories. Data was generated by using the Weka[7] data mining toolkit. Finally, a UDDI registry pointing to the WSDL definitions of the dataset services was installed on the ISISTAN cluster.

Section 4.2.1 describes the effort incurred by each Grid platform when gridifying k-NN. Section 4.2.2 evaluates performance issues.

4.2.1. Analysis of gridification effort

Certainly, measuring gridification effort and quantifying the impact of this process on the application code is difficult. Therefore, we elaborated a metric to estimate the effort when gridifying k-NN with Ibis, ProActive and JGRIM. Essentially, the estimation of the effort invested in gridification was performed by comparing the values of relevant code metrics for both the original application and its gridified counterparts. Compilation units implementing the dataset service were not considered, because the experiments were carried out as if the Grid (and therefore its services) was already established. Specifically, we employed the following code metrics [48, 49, 50]:

[7] Weka `http://www.cs.waikato.ac.nz/ml/weka`

On the Evaluation of Gridification Effort and Runtime Aspects of JGRIM Applications. (C. Mateos, A. Zunino, M. Campo). Future Generation Computer Systems: The International Journal of Grid Computing: Theory, Methods and Applications. Elsevier Science. ISSN: 0167-739X. Ed.: P. Sloot. Vol. 26, Num. 6, pp 797-819. 2010. DOI: 10.1016/j.future.2010.02.014

- *TLOC (Total Lines Of Code)* counts the total non-blank and non-commented lines across the entire application code, including the code implementing the k-NN algorithm itself, plus the code for interacting with the dataset, performing Grid exception handling and taking advantage of execution parallelization. TLOC is directly related to the extra implementation effort that is necessary to prepare the source code of an ordinary application to execute on a Grid platform.

- *PLOC (Platform-specific Lines Of Code)* counts the number of code lines that access the underlying API of the target Grid platform. Specifically, instructions pointing to API classes or invoking methods defined in these classes are computed as a PLOC line.

  The larger the value of PLOC, the more the level of tying between the application code and the Grid platform API. Clearly, it is highly desirable to keep PLOC as low as possible, so as to avoid applications to be dependent on a particular Grid platform, which in turn hinders their portability to other platforms. Intuitively, as PLOC grows, so does the time developers must spent in learning the corresponding platform API.

- *NOC/NOI (Number Of Classes/Interfaces)* represent the number of user-implemented application classes/interfaces, that is, not provided by either the Grid platform, the JVM or third-party libraries. Although simple, these metrics are useful to give an idea of the amount of object-oriented design present in the application.

- *NOT (Number Of Types)* computes the number of object types (classes and interfaces) which are defined in, and referenced from within, any of the compilation units of the application. NOT does not consider neither the Java primitive types nor the JVM bootstrap classes defined in the *java.lang* package. NOT can be viewed as the sum of NOC, NOI and the number of classes/interfaces used after the Java reserved keywords *extends*, *implements* or *import*. As a corollary, $NOT - (NOC + NOI)$ yields as a result the number of classes/interfaces not considered by the NOC/NOI metrics, that is, the object types which are defined in either the runtime system API or third-party libraries. A class which is simultaneously subclassed and imported –or similarly, an interface which is implemented and imported– is counted as *one* object type.

In order to perform a fair comparison, the following tasks were carried out on the source code of the applications before taking metrics:

- The source code was transformed to a common formatting standard, thus sentence layout was uniform across the different implementations of the application.

- Java import statements within compilation units were optimized by using the source code optimizing tool of the Eclipse IDE. This tool provides support for automatic import resolution, thus leaving in the application code only those classes/interfaces which are actually referenced by an application.

- Applications were Java 1.5 compliant, but Java generics across the source code were removed to avoid counting a line including a declaration of the form <*PlatformClass*> as a PLOC line. Otherwise, variants repeatedly using this feature across the code (e.g. in method signatures, data structure declarations, etc.) would have been unfairly resulted in greater PLOC.

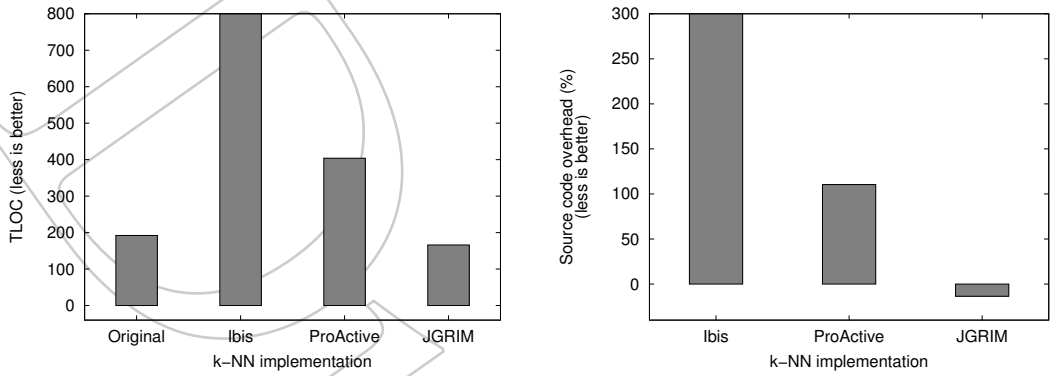
Besides, all Grid-enabled versions were implemented by the same person. The developer had very good expertise on distributed programming, and a minimal background on the facilities provided by either of the three Grid platforms. In this way, the analysis is not biased by the experience, or by different design and implementation criteria that potentially might have arisen if more than one person were involved in the gridification of the application.

Table 4 shows the resulting metrics for each implementation of the k-NN application: original, Ibis, ProActive and JGRIM. Figure 6 (a) depicts TLOC, whereas Figure 6 (b) shows the overhead in terms of extra source code lines.

On the Evaluation of Gridification Effort and Runtime Aspects of JGRIM Applications. (C. Mateos, A. Zunino, M. Campo). Future Generation Computer Systems: The International Journal of Grid Computing: Theory, Methods and Applications. Elsevier Science. ISSN: 0167-739X. Ed.: P. Sloot. Vol. 26, Num. 6, pp 797-819. 2010. DOI: 10.1016/j.future.2010.02.014

Table 4: Gridification of the k-NN algorithm: code metrics

Implementation	TLOC	PLOC	NOC	NOI	NOT	Code overhead (%)
Original	192	–	4	0	11	–
Ibis	1477	10	25	3	79	669.27
ProActive	404	11	5	**0**	37	110.42
JGRIM	**166**	**4**	**4**	2	**12**	**-13.54**



(a) TLOC (b) Source code overhead

Figure 6: TLOC and source code overhead (%) after gridification

Ibis. The size of the source code of the Ibis application was 1477 lines (seven times bigger than the original implementation). As a consequence, only a small percentage of the code resulted in pure application logic, since it was necessary to provide a lot of code mostly to implement and use a client-side proxy to the dataset Web Service. Therefore, NOC and NOT also suffered (six and seven times bigger, respectively), since more application classes and interfaces were created, and also extra APIs for low-level interaction with Web Services were imported, because Ibis does not support Web Services.

As the original k-NN application was thought to be executed on a single core machine, its classifyInstances operation was straightforwardly implemented by means of a loop control structure that iteratively feeds the classifyInstance method with the elements of the list received as an argument. To take advantage of parallelism, classifyInstances was rewritten to use the Ibis method spawning mechanism, thus each invocation to classifyInstance is concurrently executed by the Ibis platform. In this way, significant performance benefits were obtained at the cost of having an extra implementation effort since more changes to the original application were introduced. Figure 7 shows a simplified class diagram of the resulting application. Communication and coordination between the execution of classifyInstances and spawned computations was achieved by means of a *shared object* [51], a mechanism provided by Ibis to transparently share and update the state of a Java object among the distributed spawned computations of an executing application.

To further optimize the application, IbisDatasetClient was implemented to choose, upon classification of a particular instance, the service replica that is located at the cluster where the associated spawned computation is executing. The client operates as its own service broker: when data needs to be read, the client simply performs a ping to select the most appropriate WSDL description –in terms of network latency– from a list of known candidate addresses. Consequently, better interaction with the dataset was achieved. However, this mechanism forced the source code to be tied to specific dataset services, therefore lacking reusability as the information for invoking a service instance on a Grid (mostly its WSDL location) or the list of available instances for a service may vary along time. Even when this problem can be solved by employing a service registry, or alleviated by passing the list of available instances as a parameter to

On the Evaluation of Gridification Effort and Runtime Aspects of JGRIM Applications. (C. Mateos, A. Zunino, M. Campo). Future Generation Computer Systems: The International Journal of Grid Computing: Theory, Methods and Applications. Elsevier Science. ISSN: 0167-739X. Ed.: P. Sloot. Vol. 26, Num. 6, pp 797-819. 2010. DOI: 10.1016/j.future.2010.02.014

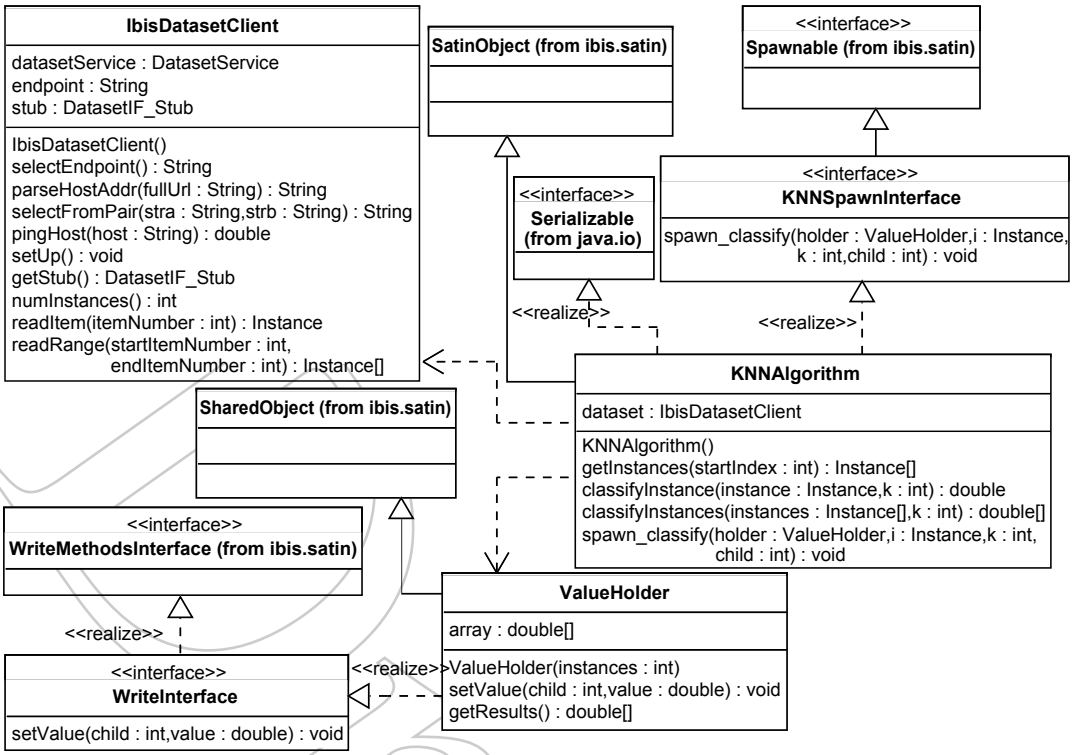


Figure 7: Class diagram of the Ibis implementation of the k-NN application

the application, the heuristic for service selection still remains hardwired in the dataset client code. In consequence, using other selection heuristic requires to reimplement/refactor this code.

ProActive. The ProActive application introduced a source code overhead near to 110%. The PLOC was slightly greater than for the Ibis case (11 lines, less than 3%) but, as indicated by NOT, the number of object types were reduced to less than a half (37 against 79). Figure 8 shows the class diagram of this application. Note that the class design is clearly much simpler than that of Ibis. However, some problems arose when gridifying with ProActive.

ProActive also lacks full support for using Web Services, as it only offers a set of classes for calling either SOAP-based services [52] or active objects. Web Service consumption within a client application is carried out by working directly with the SOAP APIs, since ProActive does not provide abstractions to support other bindings to services such as CORBA or EJB. In this sense, since all dataset replicas were initially wrapped with a non-SOAP Web Service, it was necessary to implement an active object for interfacing the data, expose it as a SOAP service, and finally write a client to use it. Directly using SOAP instead of a more generic support for Web Service invocation significantly reduced both NOC and NOT, but caused the application to be tied to a specific transport protocol for interacting with services. Furthermore, in order to allow efficient interaction between the application and the dataset, ProActiveDatasetClient was coded to employ the latency-based replica selection heuristic described before. In consequence, the dataset client shares the reusability and flexibility problems suffered by its Ibis counterpart.

Like Ibis, parallelization of classifyInstances was achieved by concurrently classifying individual instances at different Grid hosts. Specifically, a master-worker approach was followed in which, for each instance, a clone of the KNNAlgorithm class in the form of an active object is created, programmatically deployed on a particular host, and asked to perform a single classification. Whenever an active object becomes idle, another job is sent to it. Synchronization between the parent active object (i.e. the main execution thread of the application) and these clones was accomplished through the ProActive wait-by--necessity mechanism [23], which was used to block the execution of classifyInstances until any worker active object finishes its assigned job.

On the Evaluation of Gridification Effort and Runtime Aspects of JGRIM Applications. (C. Mateos, A. Zunino, M. Campo). Future Generation Computer Systems: The International Journal of Grid Computing: Theory, Methods and Applications. Elsevier Science. ISSN: 0167-739X. Ed.: P. Sloot. Vol. 26, Num. 6, pp 797-819. 2010. DOI: 10.1016/j.future.2010.02.014

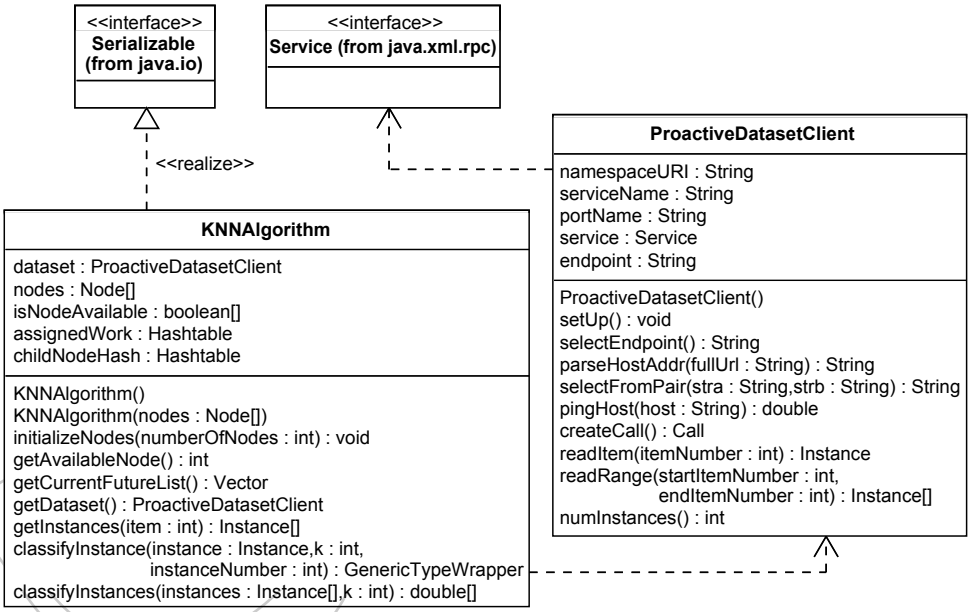


Figure 8: Class diagram of the ProActive version of the k-NN application

Sadly, some code to complement the ProActive synchronization support had to be supplied. The purpose of this code was mainly to implement behavior for keeping track of busy workers as well as assigning pending tasks to idle ones. Another alternative that was explored in an attempt to avoid this problem was to delegate job execution management to the platform. ProActive provides the so-called *technical services*, which allows developers to add non-functional concerns such as load balancing and fault tolerance to their applications without modifying their functional code [24]. However, at the time the performance experiments were performed, load balancing based on technical services was unstable.

Finally, another negative aspect that arose as a consequence of parallelization was related to the ProActive wait-by-necessity mechanism. Basically, this mechanism caused the interface of the gridified application to differ from the interface of the original implementation, because a method was changed its return type (i.e. replace the type with a ProActive API class) in order to enable the method to be called asynchronously. Particularly, the original return type of classifyInstance was modified to return instances of the GenericTypeWrapper ProActive built-in type. Consequently, the interface of the gridified application contained non-standard datatypes and interaction conventions. From the point of view of service-oriented software, this makes the interface of the ordinary version of the application no longer valid, as the gridified application does not adhere to that interface anymore. Thus, external applications (e.g. other active objects) relying on methods of k-NN have to be modified accordingly.

JGRIM. The JGRIM implementation resulted in 166 lines of code, plus 126 lines of DI-related configuration automatically generated by JGRIM (similar values were obtained for a variant using a caching policy, which is described later). The code was even smaller than the non-gridified version, since dataset access is transparently performed through discovery and invocation metaservices. Besides, exceptions caught when calling services (e.g. communication errors, timeouts, etc.) are mostly handled at the platform level and not at the application level, which helps in reducing the gridification effort and clarifying the code.

The class diagram of the JGRIM implementation of k-NN is illustrated in Figure 9. It is worth emphasizing that the tasks of extending the MFGS and PolicyAdapter classes, adding proper instance variables/setters/getters, and realizing DatasetInterface and ParallelMethodInterface were automatically performed by JGRIM. From the diagram, it can be seen that the application logic (KNNAlgorithm) does not directly reference concrete components providing Grid behavior.

The component decoupling and metaservice injection capabilities featured by JGRIM enabled to implement the interaction with the dataset service with little coding effort. Only few lines (those invoking

On the Evaluation of Gridification Effort and Runtime Aspects of JGRIM Applications. (C. Mateos, A. Zunino, M. Campo). Future Generation Computer Systems: The International Journal of Grid Computing: Theory, Methods and Applications. Elsevier Science. ISSN: 0167-739X. Ed.: P. Sloot. Vol. 26, Num. 6, pp 797-819. 2010. DOI: 10.1016/j.future.2010.02.014

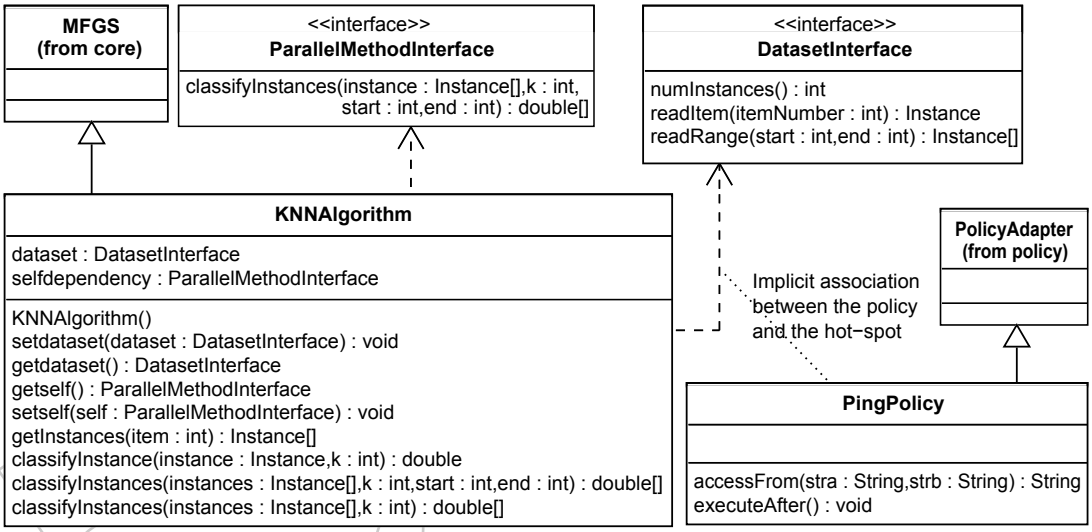


Figure 9: Class diagram of the JGRIM implementation of the k-NN application

dataset operations) were altered to use the corresponding getter, which also made the resulting code very clean and easy to understand. Moreover, introducing and testing further improvements over the algorithm outside the Grid setting is straightforward, since another implementation for the dependency to the dataset (e.g. a mock Java object) can be easily configured without modifying the application code. This is not always the case in Ibis and ProActive, since the portions of applications that are tied to Grid technologies or configuration have to be rewritten.

A policy materializing exactly the same service selection heuristic used by the Ibis and ProActive implementations was configured for the JGRIM application. Apart from its benefits in terms of flexibility and reconfigurability, an interesting aspect of using policies is that it concentrated the underlying platform API within a few classes that were external to the original application code. Besides using less API code than Ibis and ProActive (4 lines against 10/11 lines) all PLOC lines of the JGRIM application were located exclusively in the classes implementing policies, which made the application logic free from platform code. These PLOC lines were mostly calls to the profiling services of the JGRIM API, which allows developers to query the status of hardware and network resources.

The aspect of the JGRIM solution that demanded more attention from the developer was parallelism. As explained in past paragraphs, JGRIM complements self dependencies with a coordination technique that works by blocking the execution of an application the first time it reads the result of an unfinished parallel computation. As a consequence, the technique is by far less effective if an operation of a self dependency is called inside a loop control structure which accesses the result of a call before another call takes place. For example, dumbly replacing *this* by a self dependency in the original implementation of classifyInstances would have resulted in a code similar to:

```
1 public double[] classifyInstances(Instance[] instances, int k) {
2   double[] result = new double[instances.length];
3   for (int i = 0; i < instances.length; ++i){
4     double iClass = getself().classifyInstance(instances[i], k);
5     result[i] = iClass;
6   }
7   return result;
8 }
```

, which executes classification sequentially, as the computation of the class label associated to instances[i+1] do not starts until the class value of instances[i] is available. In other words, the execution remains blocked at line 5 until the computation started at line 4 calculates and assigns a value to the iClass variable.

Parallelization of classifyInstances was achieved by splitting its original implementation into two new operations: a method keeping the original interface of classifyInstances that accesses, through a self dependency, another method that actually classify a list of instances. To take advantage of the Ibis

On the Evaluation of Gridification Effort and Runtime Aspects of JGRIM Applications. (C. Mateos, A. Zunino, M. Campo). Future Generation Computer Systems: The International Journal of Grid Computing: Theory, Methods and Applications. Elsevier Science. ISSN: 0167-739X. Ed.: P. Sloot. Vol. 26, Num. 6, pp 797-819. 2010. DOI: 10.1016/j.future.2010.02.014

parallelization services, the latter method was implemented in a recursive way by following the Ibis coding conventions. The resulting code was:

```
public double[] classifyInstances(Instance[] instances, int k){
  return getself().classifyInstances(instances, k, 0, instances.length);
}

public double[] classifyInstances(Instance[] instances, int k,
                                  int start, int end){
  if (end - start == 1){
    double iClass = classifyInstance(instances[start], k);
    return new double[] {iClass};
  }
  int mid = (start + end) / 2;
  double[] left = classifyInstances(instances, k, start, mid);
  double[] right = classifyInstances(instances, k, mid, end);
  double[] result = new double[left.length + right.length];
  System.arraycopy(left, 0, result, 0, left.length);
  System.arraycopy(right, 0, result, left.length, right.length);
  return result;
}
```

In this way, the execution of classifyInstances(Instance[], int, int, int) is delegated to an external execution service, in this case Ibis, and the signature of the original classifyInstances method is maintained. Therefore, the resulting MFGS exposes the same service interface as the original application. Additionally, note that the transformation did not require to use any Grid API code at all. Finally, many simple methods can be found in the literature to manually convert from iterative to recursive code, and vice versa.

The situation described before is an example of a common tradeoff in parallel programming: independence from programming APIs versus flexibility to control application execution [53]. At the programming language level, the approaches to parallel processing can be classified into implicit or explicit. Implicit parallelism allows programmers to write their programs without any concern about the exploitation of parallelism, which is instead automatically performed by the runtime system. Conversely, languages based on explicit parallelism offer synchronization/coordination primitives for describing the way parallel computations take place. The programmer has absolute control over parallel execution, thus it is feasible to take advantage of parallelism to implement very efficient applications. However, programming with explicit parallelism is more difficult, since the burden of initiating, stopping and synchronizing parallel executions is placed on the programmer.

Platforms like ProActive and to a lesser extent Ibis are designed to provide explicit parallelization, as they are performance-oriented. The programmer has finer control of parallelism, but gridified applications are more difficult to understand and to maintain. JGRIM promotes implicit parallelism, which does not require to explicitly use extra API code. In other words, JGRIM not only takes care of performance, but also pays attention to portability, maintainability and legibility of gridified code.

Discussion. An interesting result of the evaluation is the size of the compiled versions of the different k-NN implementations. Table 5 shows bytecode information, given by the number of generated .class files and the total size (in Bytes) of these files after compilation, and plugging information, given by the number of total lines of code (and PLOC lines) destined to run the applications on our Grid.

Table 5: Characteristics of the k-NN implementations upon execution on the Grid setting

Implementation	# of .class	Bytecode size	Bytecode overhead (%)	Plugging lines (regular/PLOC)
Original	4	8242	–	–
Ibis	45	111452	1252	43/0
ProActive	5	17750	115	72/15
JGRIM	13	32028	288	48/4

On the Evaluation of Gridification Effort and Runtime Aspects of JGRIM Applications. (C. Mateos, A. Zunino, M. Campo). Future Generation Computer Systems: The International Journal of Grid Computing: Theory, Methods and Applications. Elsevier Science. ISSN: 0167-739X. Ed.: P. Sloot. Vol. 26, Num. 6, pp 797-819. 2010. DOI: 10.1016/j.future.2010.02.014

After source code gridification and compilation, the binary size of the ProActive implementation was about 17 KB, versus 31 KB and 108 KB of the JGRIM and Ibis, respectively. In this latter case, though much functionality for interacting with the Web Service dataset was added to the original application (resulting in approximately 74 KB of bytecode), the final deployment generated a lot of .class files for supporting parallelism, managing shared objects and carrying out platform-specific object serialization, therefore increasing the amount of bytecode of the whole application. Thus, transferring the code for execution on a remote host will require more bandwidth than either the ProActive and JGRIM implementations. At present, the Ibis platform does not support automatic bytecode transfer for applications.

The bytecode of the ProActive solution was about a half of the bytecode of the JGRIM application. However, it is worth noting that ProActive dynamically adds mobility to applications by enhancing their bytecode not at deployment time, but at runtime, thus this overhead is not present in the above results because it is very difficult to measure. On the other hand, it was determined that a big percentage of bytecode for the JGRIM implementation were instructions for dealing with thread-level serialization and migration, and bridging the application with the Ibis services, this latter representing a 72% of the total bytecode. In fact, the binary size without this support was even smaller than the compiled version of the original k-NN implementation. In this sense, to make the binary code lighter and more compact, at least in appearance, a technique for instrumenting bytecode similar to the one used by ProActive could be implemented. Basically, the idea is to develop a special Java class loader that instruments applications at runtime, thus dynamically enabling them for being parallel as well as mobile. In this way, dynamic code transfer is more efficient: when a host does not have the necessary bytecode to execute a JGRIM application, the binary code that is transferred to it is just the non-instrumented version, thus saving network bandwidth. In addition, a mechanism for caching instrumented classes could be employed so as to avoid instrumenting the same application many times.

Table 5 also includes the amount of source code that was implemented to execute the applications, and how much of this code was concerned with accessing the underlying platform API. To a certain extent, the amount of implemented lines can be interpreted as an indicator of the effort demanded by either of those platforms to execute a gridified application onto the Grid. Quantifying this effort is important since a major goal of Grids is to allow users to run applications in a *plug and play* fashion. Taking into account that learning a Grid API is indeed a time-consuming task, the effort can be approximated by the following formula:

$$PlugEffort = (PlugLines - PLOC_{Plug}) + PLOC_{Plug} * APIFactor$$

where $APIFactor$ is a numeric value that represents the complexity of the platform API being used (i.e. ProActive, Ibis or JGRIM). The formula adjusts the lines representing Grid code to incorporate the effort invested by a programmer in learning these APIs. As here defined, $APIFactor$ is highly influenced by how much the programmer knows about a particular Grid API and its associated abstractions before performing gridification. In our experiments, as the k-NN application was gridified by only one person who did not have a solid knowledge about any of the platforms, this influence is not present. In addition, since the developer had a good background on distributed and parallel programming, the difficulty in learning each API was similar. Hence, we can assume that $APIFactor_{Ibis} = APIFactor_{ProActive} = APIFactor_{JGRIM}$, which means the developer spent almost the same time to learn each one of the three APIs.

The above formula is a rough approximation to truly quantifying the necessary effort to execute a gridified application on a Grid. Certainly, many aspects intimately related to Grid application execution and deployment (e.g. creating/editing configuration files, performing network-specific settings, initiating the execution of the application itself, and so on) are obviously out of the scope of this formula. However, as the purpose of this article is not to measure gridification effort beyond implementation code, the formula is a good approximation.

We can extend this idea to take into account the code metrics reported previously to obtain an estimation of the effort incurred by each platform in gridifying an application. Two fundamental aspects that characterize the existing gridification tools are concerned with how much redesign and code modification (within compilation units) they impose on input applications [12]. Hence, the overall gridification effort can be thought as composed of three different effort factors: restructuring the application (e.g. merging or splitting components), adapting the code of its individual compilation units, and plugging the resulting application into a Grid (modeled by $PlugEffort$).

On the Evaluation of Gridification Effort and Runtime Aspects of JGRIM Applications. (C. Mateos, A. Zunino, M. Campo). Future Generation Computer Systems: The International Journal of Grid Computing: Theory, Methods and Applications. Elsevier Science. ISSN: 0167-739X. Ed.: P. Sloot. Vol. 26, Num. 6, pp 797-819. 2010. DOI: 10.1016/j.future.2010.02.014

Since the implementation language of the original k-NN application and the gridified applications is the same (Java), we can obtain an estimation of the effort, in terms of source code lines, necessary to carry out this transformation, plus the effort to put the application to work, by the formula:

$$GE\,(GridificationEffort) = ReimplEffort + RedesignEffort + PlugEffort$$

where:

$$ReimplEffort = |TLOC_{Grid} - TLOC_{Orig}| + PLOC * APIFactor$$

$$PlugEffort = (PlugLines - PLOC_{Plug}) + PLOC_{Plug} * APIFactor_{Plug}$$

ReimplEffort is computed as the difference between the amount of source code lines of the original and the gridified application, plus the adjustment of PLOC lines. If the difference is positive, extra lines to the original implementation are added, whereas a negative difference indicates that the size of the new application is smaller. To model these cases with a single expression, we assume that adding code lines is as laborious as removing lines from the original application. As ProActive, Ibis and JGRIM focus on gridifying by modifying the compilation units of an application but not its structure, $RedesignEffort$ was assumed to be zero.

Intuitively, *APIFactor* at implementation time should always be greater than the one included in the computation of $PlugEffort$, because at plugging time the developer is likely to be more familiarized with the Grid programming API. In our experiments, this rule does not hold since the portions of the platform APIs (i.e. object types) that were used when performing both steps were disjoint. Then, *APIFactor* at implementation and plugging time were the same, and they were both set to 30 (i.e. a PLOC line is worth 30 regular code lines). Although this factor was determined arbitrarily, in the future it may be obtained from experiences in gridifying similar applications.

As the gridification processes of Ibis, ProActive and JGRIM produce Java applications, comparison of the *GE* values resulted from gridifying the original k-NN application (also written in Java) is possible. To further simplify the comparison and to better perceive the relative differences between the computed values, *GE* was normalized according to the following formula:

$$NormalizedGE = \frac{GE}{ScalingFactor}$$

where:

$$ScalingFactor = 10^{truncate(log_{10}[max(GE_{Ibis}, GE_{ProActive}, GE_{JGRIM})])}$$

Computing *NormalizedGE* on the gridified versions of the k-NN algorithm resulted in 1.63 (Ibis), 1.05 (ProActive) and 0.31 (JGRIM) (see Figure 10). Roughly speaking, *GE* suggests that Ibis demanded more effort from the developer than both ProActive and JGRIM, which decreased Ibis effort by 36% and 81%, respectively. It is worth emphasizing that these results cannot be generalised to other applications, since they merely represent an indicator of how much effort each platform demanded to gridify the original k-NN implementation taking into account the assumptions explained in previous paragraphs.

4.2.2. Analysis of performance and network usage

To evaluate the runtime behavior of the applications with respect to response time and network resource usage, each gridified version of the algorithm was employed to classify several list of instances with different sizes. Network resource consumption (generated TCP traffic and amount of data packets) was measured by using the *tcpdump* network monitoring program[8] and then analyzed with the Wireshark[9] software. Loopback network traffic was filtered out.

[8] Tcpdump/libpcap `http://www.tcpdump.org`

[9] The Wireshark Network Protocol Analyzer `http://www.wireshark.org`

On the Evaluation of Gridification Effort and Runtime Aspects of JGRIM Applications. (C. Mateos, A. Zunino, M. Campo). Future Generation Computer Systems: The International Journal of Grid Computing: Theory, Methods and Applications. Elsevier Science. ISSN: 0167-739X. Ed.: P. Sloot. Vol. 26, Num. 6, pp 797-819. 2010. DOI: 10.1016/j.future.2010.02.014

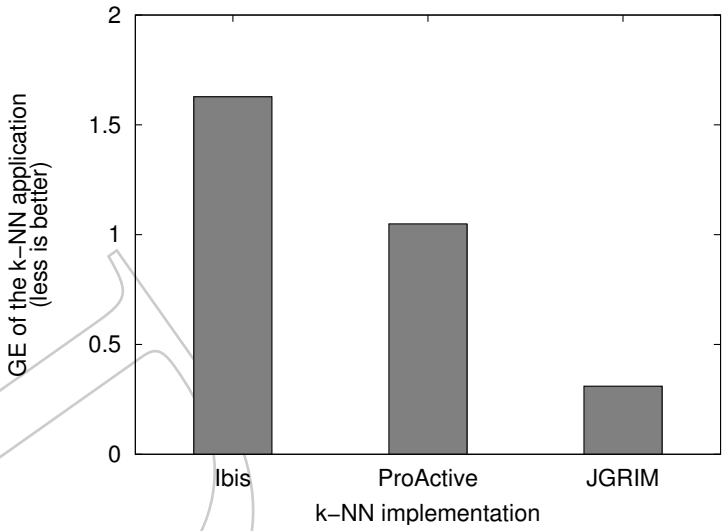


Figure 10: Gridification effort for the k-NN application

Furthermore, all tests were started in one Grid machine, thus launching conditions were exactly the same. In addition, since Ibis does not support dynamic transfer of application bytecode, the executable codes of the gridified applications were manually copied to each host of the Grid. Besides, Ibis and JGRIM used the Satin cluster-aware random stealing algorithm [18] for scheduling parallel computations. These tasks, which enabled a fair evaluation of the runtime behavior of the gridified k-NN applications, were also carried out when experimenting with the restoration application.

Each test battery performed on a gridified application involved ten executions of the classification algorithm on input lists composed of 5, 10, 15, 20 and 25 instances. The resulting execution times were averaged to compute the total execution time (TET). On the other hand, for practical reasons, network resource consumption was measured by taking into account the total amount of network traffic and packets generated during an entire test battery.

Comparison of TET. The TET obtained from the different Grid-enabled implementations of k-NN is shown in Figure 11. Particularly, the figure shows the average response time for Ibis, ProActive, JGRIM, and a variant of JGRIM using a policy that, besides selecting services based on network bandwidth, stored in an in-memory local cache the data read from the dataset. Caches were configured to have an unlimited size (big enough to hold the entire dataset) and non-volatile entries, thus cached information persisted across the executions of individual test batteries. Consequently, dataset replicas were accessed completely from each Grid host only once per test battery. The implementation of the caching policy was straightforward (13 lines of code), and explicitly used only two operations of the JGRIM policy API. The figure also shows errorbars in the *y* axis corresponding to the standard deviation in each case.

The ProActive application experienced lower performance with respect to the Ibis application and the plain version of JGRIM (i.e. not using the caching policy). A considerable percentage of the time was spent by ProActive in remotely creating JVMs on every Grid node before executing the application, which demanded on average 40 seconds. Indeed, easy deployment of Grid applications is one of the good features of ProActive. However, the results show that this feature conditioned the overall performance of the k-NN application. In principle, the results suggest that ProActive may not be suitable for running computations whose response time is similar or slightly greater than the time required to remotely initialize the ProActive runtime system on a Grid.

The Ibis application performed better than the JGRIM application. Specifically, the JGRIM implementation added a performance overhead in the range of 10-20%. However, much of this time (20 seconds on average) corresponded to querying the UDDI registry. Despite the obvious overhead of service brokering, it allows the application code to be completely isolated from the actual service instances implementing a certain functionality, which are instead discovered at runtime by the JGRIM platform. In consequence,

On the Evaluation of Gridification Effort and Runtime Aspects of JGRIM Applications. (C. Mateos, A. Zunino, M. Campo). Future Generation Computer Systems: The International Journal of Grid Computing: Theory, Methods and Applications. Elsevier Science. ISSN: 0167-739X. Ed.: P. Sloot. Vol. 26, Num. 6, pp 797-819. 2010. DOI: 10.1016/j.future.2010.02.014

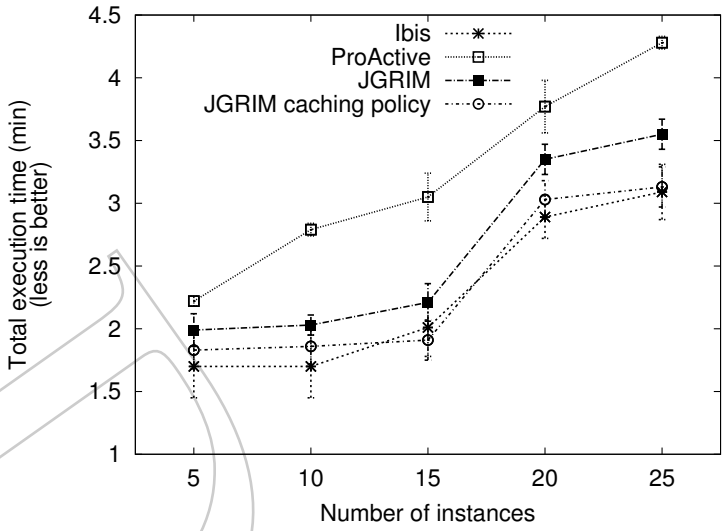


Figure 11: TET (min) of the k-NN application

the implementation code is shorter and cleaner, and remains free from instructions for finding Grid services. Furthermore, brokering enables for a better reuse of services, since applications and Grid services are bound in a dynamic, more flexible way. For example, executing applications can discover new services as they are published. It is worth mentioning that brokering capabilities in JGRIM can be disabled without modifying the application code by simply editing external configuration. In this way, better performance can be potentially obtained, at the cost of underexploiting the available Grid services.

To keep the JGRIM application using the UDDI registry and at the same time decrease its response time, a simple caching mechanism was implemented through a policy. This policy was basically an extension of the policy for selecting the nearest dataset service replica. As depicted in Figure 11, the alternative JGRIM application reduced the TET of the non-cached JGRIM implementation by 8-16%, and achieved performance levels similar to the Ibis implementation.

The weak point of the solution is that it increased the total memory usage within the Grid by about 100 MB, that is, the extra cache memory that was allocated across hosts to store objects representing the instances of the dataset. In production Grids, where many different applications compete for the available resources, allocating memory and storage resources at will may be disallowed. However, the goal here was to show the flexibility of policies to effectively and non-invasively tune JGRIM applications. Note that both the ProActive and Ibis applications might have benefited from the same caching mechanism, but this would have lead to introducing yet more modifications to the original application code, thus increasing GE.

In parallel computing, speedup refers to how much a parallel algorithm is faster than its sequential counterpart [54]. Speedup is defined as T_1/T_p, where T_1 is the execution time of the sequential algorithm, and T_p represents the execution time of the parallel version of the algorithm on p CPUs. Figure 12 depicts the speedups associated to the k-NN application, this is, the time necessary to execute the original implementation, which sequentially classifies instances based on a file-based dataset, over the time required to run the Grid-enabled implementations, which classify instances in parallel based on a dataset service. All tests corresponding to the original application were run on "VPNServer", which is a fast machine.

Note that, despite achieving different speedup levels, the speedup curves of the Ibis and the two variants of JGRIM seemed to have the same behavior. Basically, this is because these implementations share the same execution model (i.e. the scheduling mechanism of Satin) to classify individual instances in parallel, which is the stage of the whole classification process where applications spent most of the time. ProActive appeared to gain efficiency as the number of instances increased, but more experiments should be conducted to corroborate this trend. Overall, the implications of the resulting speedups are twofold. First, the original application certainly benefited from being ported to the Grid. Second, and more important, JGRIM achieved speedups levels that are similar to those achieved by Ibis and ProActive.

On the Evaluation of Gridification Effort and Runtime Aspects of JGRIM Applications. (C. Mateos, A. Zunino, M. Campo). Future Generation Computer Systems: The International Journal of Grid Computing: Theory, Methods and Applications. Elsevier Science. ISSN: 0167-739X. Ed.: P. Sloot. Vol. 26, Num. 6, pp 797-819. 2010. DOI: 10.1016/j.future.2010.02.014

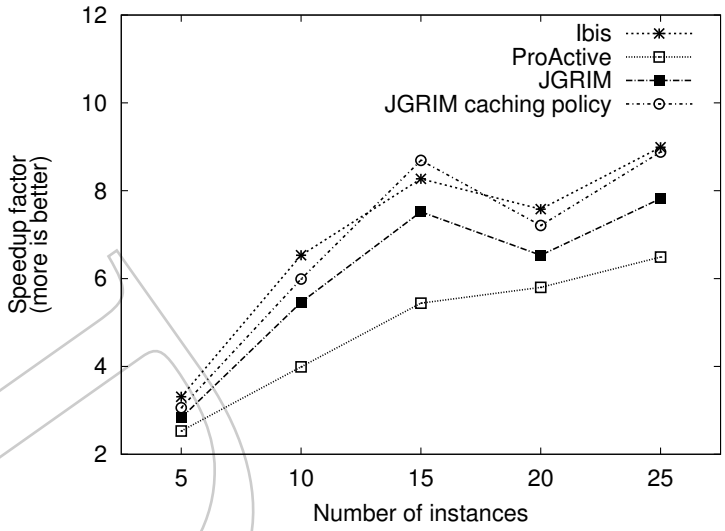


Figure 12: Speedup introduced by the gridified versions of k-NN

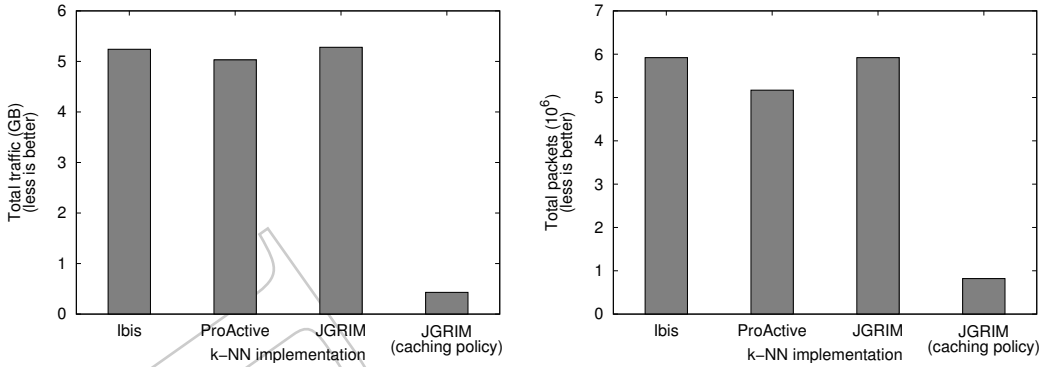
Comparison of network traffic. Figure 13 (a) illustrates the total network traffic (measured in GB) that was generated across the Grid for ten runs of each gridified implementation of k-NN. This traffic includes the amount of data that was sent from any of the Grid machines to another one residing in the same or other cluster. Although LAN communication is significantly cheaper than communication between Internet--connected machines, the traffic associated to intra and extra cluster communication was not discriminated because the latter just represented a very small fraction of the total traffic. In other words, even when the Grid-enabled applications are service-bound (i.e. they perform a large number of accesses to the dataset service), each cluster hosted a replica of the dataset thus service requests were always performed through local network links.

Concretely, the traffic destined to extra cluster communication was only 23.46 MB (Ibis), 12.98 MB (ProActive), 30.06 MB (JGRIM, without caching), and 26.95 MB (JGRIM, with caching). The relation between these values clearly cannot be generalised, but they are certainly a consequence of the way each tool manages the execution of applications at the platform level. In Ibis, idle machines –that is, machines not executing a spawned computation– periodically generate requests to other nodes of the Grid participating in the execution of an application to get an unfinished computation from those nodes. Of course, requests can be sent to local as well as remote nodes. JGRIM applications inherit this behavior since parallelization is based on Satin/Ibis. Then, extra cluster traffic in JGRIM comprises the traffic generated when sending steal requests, querying UDDI and propagating profiling information. On the other hand, instead of employing this pull approach to job execution, the ProActive application used a push approach to send execution request to Grid machines only when they became idle. In this way, less extra cluster communication was generated. However, the application code resulted in a mix of logic code and instructions to manage the classification of individual instances on specific nodes.

The plain JGRIM implementation added an overhead of less than 1% to the total traffic generated by the Ibis implementation, which represented about 4 MB of extra traffic per run in a test battery. Note that the overhead is acceptable because part of this traffic was destined to support Grid service brokering, whose benefits have been already discussed. Furthermore, the traffic related to the diffusion of profiling information gradually spread over the entire execution time, which, unlike bursty communication, do not causes communication bottlenecks. In short, these results show that the JGRIM solution did not incur in high communication overheads with respect to the application directly using the Ibis runtime.

The JGRIM application with the caching policy drastically decreased the total traffic. Particularly, it reduced the traffic generated by the plain JGRIM implementation by 92%. Obviously, these gains are a direct consequence of reducing the number of accesses to the dataset service. Interestingly, the caching mechanism was implemented without altering the structure or the logic of the application code. Again,

On the Evaluation of Gridification Effort and Runtime Aspects of JGRIM Applications. (C. Mateos, A. Zunino, M. Campo). Future Generation Computer Systems: The International Journal of Grid Computing: Theory, Methods and Applications. Elsevier Science. ISSN: 0167-739X. Ed.: P. Sloot. Vol. 26, Num. 6, pp 797-819. 2010. DOI: 10.1016/j.future.2010.02.014



(a) Total traffic (b) TCP packets

Figure 13: Total traffic (GB) and amount of TCP packets generated during each test battery

similar improvements might have been added to the Ibis and ProActive solutions, but this would have required to alter the code of the dataset client.

Figure 13 (b) illustrates the amount of network packets generated by each variant of the k-NN application during an entire test battery. The amount of packets generated by the Ibis and the plain JGRIM variants were very similar. On the other hand, for reasons that are similar to that of network traffic, the ProActive solution outperformed both the Ibis and JGRIM implementations. Finally, the JGRIM version using the caching policy reduced the amount of packets generated by ProActive by 84%.

4.3. Panoramic image restoration

This section reports the experiments with the application for image restoration discussed in Section 3.3. Specifically, Section 4.3.1 analyzes gridification effort. Section 4.3.2 describes the experiments related to performance and usage of network resources.

4.3.1. Analysis of gridification effort

To evaluate the characteristics of the gridified versions of the application, the same code metrics of Section 4.2.1 were employed. Besides, we used a modified version of the application presented in Section 3.3 that processed input images by using several workers. Each worker was initially implemented as a thread that communicated results to a master thread by means of explicit object referencing. Enhancement of individual portions of the image was implemented based on the algorithm proposed in [55]. Table 6 shows the resulting code metrics for the Ibis, ProActive and JGRIM implementations of this application.

Table 6: Gridification of the restoration application: code metrics

Implementation	TLOC	PLOC	NOC	NOI	NOT	Code overhead (%)
Original	241	–	3	1	37	–
Ibis	227	5	**3**	1	40	-5.81
ProActive	299	17	4	1	46	24
JGRIM	**226**	**0**	**3**	1	**36**	**-6.22**

As illustrated in the table, the size of the applications did not vary significantly among each other. The ProActive version introduced a code overhead of 24%, while the Ibis and JGRIM implementations reduced the size of the original code by about 6%. Unfortunately, the negative overhead resulted from removing source code lines implementing critical portions of the non-gridified application, namely, code implementing synchronization and coordination behavior between the master and its workers. In opposition, the ProActive application preserved the interaction scheme originally designed to communicate and

On the Evaluation of Gridification Effort and Runtime Aspects of JGRIM Applications. (C. Mateos, A. Zunino, M. Campo). Future Generation Computer Systems: The International Journal of Grid Computing: Theory, Methods and Applications. Elsevier Science. ISSN: 0167-739X. Ed.: P. Sloot. Vol. 26, Num. 6, pp 797-819. 2010. DOI: 10.1016/j.future.2010.02.014

coordinate the master and its workers. In fact, one of the first official uses of ProActive was precisely master-worker Grid applications [56]. Nevertheless, Ibis and JGRIM solutions were also able to provide a fitting alternative to coordinate these components.

Master and workers in ProActive were implemented as active objects. ProActive provides a mechanism so that references between active objects are transparently managed by means of specialized proxies that hide the physical location of active objects. When a method call is performed on an active object A, the invocation is transparently forwarded to the actual object instance representing A, regardless where it is currently located. In consequence, worker instances in the ProActive application were easily implemented to send their results to the master active object, to which a reference is obtained when a worker is first created and transparently maintained when the worker is migrated. However, while ProActive greatly preserved the interaction mechanism of the original application components (i.e. explicit references between master and workers), some code using API calls had to be supplied to implement active object deployment and task management, which in turn affected TLOC and PLOC.

In contrast, unless explicitly managed by means of its communication API, Ibis does not transparently maintain references upon distribution of application objects. As a consequence, spawning a computation W from within a method of a master M which passes itself as an argument to W results in losing the reference to M in W. Another alternative that was explored, similar to the k-NN Ibis implementation, involved the use of a shared object to indirectly communicate the master and the workers. Unlike the k-NN application, in which the output of spawned computations are just numerical values, the results coming from workers are processed subimages that may be large in size. Since Ibis automatically broadcasts an individual write to a shared object to its distributed copies, employing this mechanism would had required too much network resources. This can be avoided by maintaining a list of shared objects, each handling the communication of the master and exactly one worker. However, this would had made the implementation of the application more difficult.

To overcome this situation, the iterative-like work submitting structure of the original application was transformed to a recursive algorithm, in which each leaf of the execution tree represents a computation in charge of processing an individual portion of the image being restored. The code structure was similar to the one presented in Section 4.2.1. A code snippet of the master component is presented next:

```
public void restoreSubImages(Vector tiles){
  ...
  Vector results = restoreSubImages(tiles, 0);
  // Ibis synchronization primitive
  super.sync();
  ...
}

public Vector restoreSubImages(Vector tiles, int currentChild){
  if (tiles.size() == 1){
    byte[] subImagePixels = (byte[]) tiles.firstElement();
    Hashtable data = new Hashtable();
    data.put("pixels", subImagePixels);
    data.put("iterations", iterations);
    Worker worker = new Worker(data, currentChild);
    Vector result = new Vector();
    result.addElement(worker.run());
    return result;
  }
  int mid = tiles.size()/2;
  Vector tilesStart = splitTiles(tiles, 0, mid);
  Vector tilesEnd = splitTiles(tiles, mid, tiles.size());
  // Spawned computations
  Vector left = restoreSubImages(tilesStart, currentChild);
  Vector right = restoreSubImages(tilesEnd, currentChild + mid);
  // Ibis synchronization primitive
  super.sync();
  left.addAll(right);
  return left;
}
```

The JGRIM implementation demanded some modifications that can be grouped in two categories.

On the Evaluation of Gridification Effort and Runtime Aspects of JGRIM Applications. (C. Mateos, A. Zunino, M. Campo). Future Generation Computer Systems: The International Journal of Grid Computing: Theory, Methods and Applications. Elsevier Science. ISSN: 0167-739X. Ed.: P. Sloot. Vol. 26, Num. 6, pp 797-819. 2010. DOI: 10.1016/j.future.2010.02.014

On one hand, similar to its Ibis counterpart, a recursive restoreSubImages method was implemented to take advantage, through a self dependency, of the execution and parallelization services provided by the Ibis platform. Besides for leveraging the Ibis services, the transformation was necessary since JGRIM discourages explicit referencing between application components (in this case master and workers) because in practice tight coupling leads to poor reusability and portability of component code. In the end, the master component of the JGRIM application *indirectly* used workers through a self dependency.

On the other hand, the dependency from the master component (implemented by the ImageRestorer class) to the FTP downloader component was identified as a hot-spot. In consequence, direct accesses to this component were replaced across the source code of ImageRestorer by calls to the corresponding getdownloader method. As in the case of the k-NN application and the dataset component, this replacement enables the use of other components for downloading images implementing different protocols (e.g. HTTP, Web Services, GridFTP, and so on) without modifying the application logic. Moreover, a simple policy to move the gridified application to the image repository location upon the initial attempt to download an input image was associated to this hot-spot:

```
import jgrim.core.MFGS;

public class AlwaysMovePolicy extends jgrim.policy.PolicyAdapter{
  public void executeBefore(){
      MFGS app = (MFGS)getExecContext().getSourceComponent();
      app.moveTo("VPNServer");
  }
}
```

In contrast, this kind of performance improvement could not be introduced in the Ibis implementation, since Ibis implicitly manages migration of spawned computations between machines and does not let applications to explicitly control this feature. Moreover, ProActive do provide a migrateTo primitive for moving active objects. However, this primitive implements a weak migration mechanism [57], thus placing a burden on the developer since inherently complex and lengthy code to manually capture and resume the execution state of applications must be supplied. It was decided not to add mobility to the ProActive application so as to fairly keep its TLOC and PLOC values low.

Table 7 shows the bytecode information associated to the different versions of the restoration application, including a simple application launcher. Since the restoration application required less Grid functionality than the k-NN application (i.e. parallelism but not FGSs), bytecode information can be used to get a more accurate estimation of the minimum bytecode overhead introduced by each platform to any application. Since ProActive instruments applications at runtime, the bytecode overhead after compiling the gridified application was less than the one introduced by Ibis and JGRIM. Furthermore, the JGRIM implementation introduced a bytecode overhead of 15% compared to the Ibis solution, which arose as a consequence of enabling the application for being mobile and using the Satin parallelization support. This is an undesirable nevertheless acceptable overhead given the added value of JGRIM applications in terms of component reusability and decoupling, and the possibility of leveraging the execution services provided by other Grid platforms. In any case, dynamic class instrumentation could be employed to cut down some of this overhead.

Table 7: Characteristics of the restoration applications upon execution on the Grid setting

Implementation	# .class	Bytecode size	Bytecode overhead (%)	Plugging lines (regular/PLOC)
Original	4	12731	–	–
Ibis	10	30515	139	32/0
ProActive	5	16852	32	48/11
JGRIM	11	35137	175	23/7

Figure 14 shows the resulting *GE* for the different versions of the image restoration application. To make these results comparable to the ones described in Section 4.2.1, *GE* values were scaled down by

On the Evaluation of Gridification Effort and Runtime Aspects of JGRIM Applications. (C. Mateos, A. Zunino, M. Campo). Future Generation Computer Systems: The International Journal of Grid Computing: Theory, Methods and Applications. Elsevier Science. ISSN: 0167-739X. Ed.: P. Sloot. Vol. 26, Num. 6, pp 797-819. 2010. DOI: 10.1016/j.future.2010.02.014

using the same factor (10^3). According to *GE*, the restoration application was easier to gridify than the k-NN application. This situation truly makes sense because the gridification of the former application required less Grid services (i.e. only parallelism). Specifically, functional Grid services were not used, and less policy code was implemented when gridifying with JGRIM.

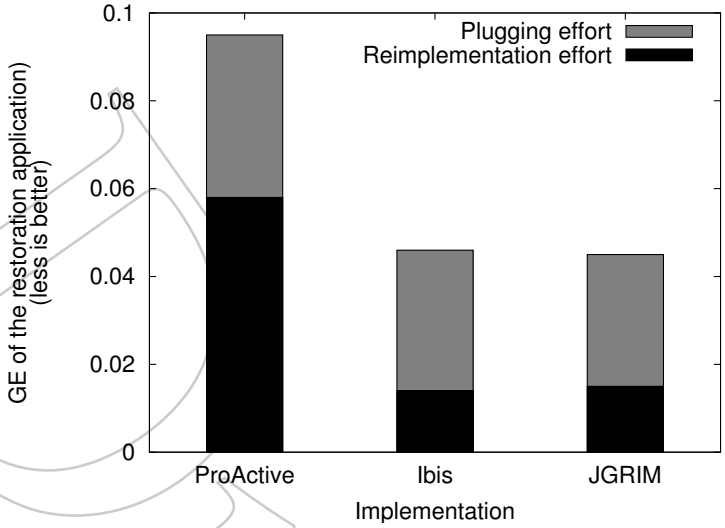


Figure 14: Gridification effort for the restoration application

The resulting *GE* was 0.046 (Ibis), 0.095 (ProActive) and 0.045 (JGRIM), which indicates that, for the case of this application, gridifying with ProActive demanded more effort than doing so with Ibis or JGRIM. Specifically, Ibis reduced *GE* of ProActive by about 51.5%, whereas JGRIM decreased this effort by about 52.6%. As illustrated in the figure, most of the effort when gridifying with Ibis and JGRIM was invested in providing code to launch the execution of the Grid-enabled application. In any case, the plugging effort can be further reduced by providing better tools to easily run gridified applications. In fact, one important feature that is currently missing in JGRIM, but will be supplied in the near future, is concerned with the provision of tools to graphically launch and monitor the execution of gridified applications.

As observed, the reimplementation effort in JGRIM was slightly greater than in Ibis. However, one important point must be clarified. The restoration application is inherently a pure parallel application, and is not intended to take advantage of other Grid resources or services but distributed processors. These characteristics makes the parallelization services of Ibis an appropriate support for implementing the application. Furthermore, JGRIM is a gridification tool whose goal is to isolate applications from specific Grid services and therefore how parallelization is performed, which in the experiments translated in a little extra effort at gridification time. Nevertheless, this small overhead is acceptable given the benefits of JGRIM in terms of maintainability and portability of gridified code. The logic of the JGRIM restoration application resulted absolutely clean of Grid API code and was decoupled from concrete parallelization services, thus workers can be configured to be deployed on other execution services (e.g. threads, Globus jobs, ProActive active objects, etc.) without source code modification. In this sense, existing Grid platforms like ProActive or Ibis and our work complement each other.

Since the k-NN and the restoration application were gridified by the same person, and this former was ported first to our Grid, it is clear that the developer had knowledge about each Grid API at the time of gridifying the latter application. In this light, *GE* was computed by using $APIFactor = 1$ (i.e. a PLOC line is worth one regular code line). The goal of this adjustment is to model the fact that programmers often face a steep learning curve when employing new Grid technologies, but as they use them, the learning effort tends to disappear. Learning curves corresponding to the three platforms used in the experiments were assumed to be very close to each other because the developer had a strong background on distributed and parallel programming, the complexity of the Grid APIs was similar, and the programming language was Java.

On the Evaluation of Gridification Effort and Runtime Aspects of JGRIM Applications. (C. Mateos, A. Zunino, M. Campo). Future Generation Computer Systems: The International Journal of Grid Computing: Theory, Methods and Applications. Elsevier Science. ISSN: 0167-739X. Ed.: P. Sloot. Vol. 26, Num. 6, pp 797-819. 2010. DOI: 10.1016/j.future.2010.02.014

4.3.2. Analysis of performance and network usage

This section reports the performance tests that were conducted based on the various implementations of the restoration application. To better understand the runtime behavior of these applications, and set the basis for a more detailed comparison, a simple benchmark based on the original application was prepared. To this end, we installed an image repository (FTP server) on the "VPNServer" machine at the ISISTAN cluster. The repository contained an RGB image in JPEG format of 2408.18 KB (10120 x 2200), and four more pictures obtained from rescaling this image down to 76, 63, 37 and 18 percent. Rescaled images resulted in a size of about 1.8 MB (9100 x 1980), 1.5 MB (8080 x 1760), 900 KB (6060 x 1320) and 400 KB (4040 x 880), respectively.

The purpose of the benchmark was to have an estimation of the average time necessary to download and restore an image from the repository. These two factors were estimated separately. On one hand, the average transfer time was measured by performing five different downloads of each image from the remote "ale" machine residing at the Ale cluster. To avoid unnecessary noise, all tests were run under low network load. On the other hand, the average execution time required to enhance an image was estimated by averaging five runs of the restoration process on a random horizontal portion of a size of 1/20 of the above images, and multiplying the elapsed time by 20. These tests were run on the "VPNServer" machine.

The experiments using the gridified versions of the application were performed by processing each image from the repository ten times, and then computing the resulting TET (in minutes) and throughput (in KB restored per minute). Furthermore, images were restored by splitting them into 20 subimages. As in the case of the k-NN application, network resources consumed by applications, namely the generated TCP traffic (in MB) and the amount of data packets, were measured by using the tcpdump and Wireshark tools. These tests were initiated in the "ale" machine.

Comparison of TET. The TET values of the gridified applications are shown in Figure 15. Specifically, the figure illustrates the execution performance of the Ibis, ProActive and JGRIM implementations, and the variant of JGRIM employing the policy that moves the application to the image repository location upon downloading an individual image. As explained, the implementation of the policy was easy, because only involved the use of two lines of code for moving the MFGS to "VPNServer". The graphic includes errorbars in the *y* axis corresponding to the standard deviation of the elapsed times.

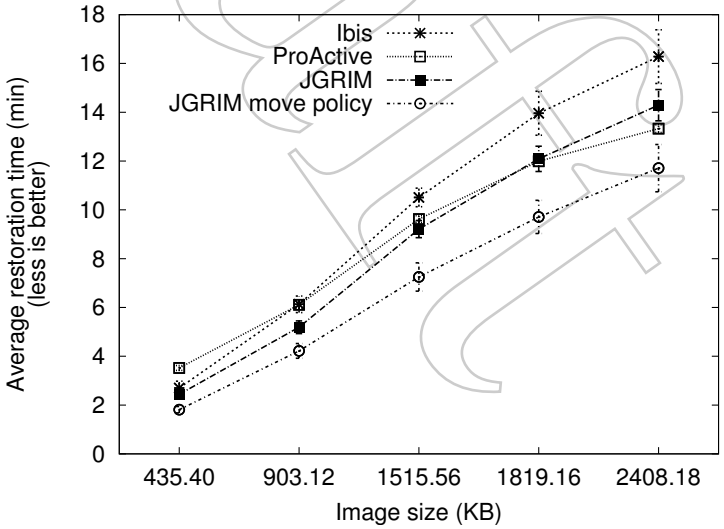


Figure 15: TET (min) of the restoration application

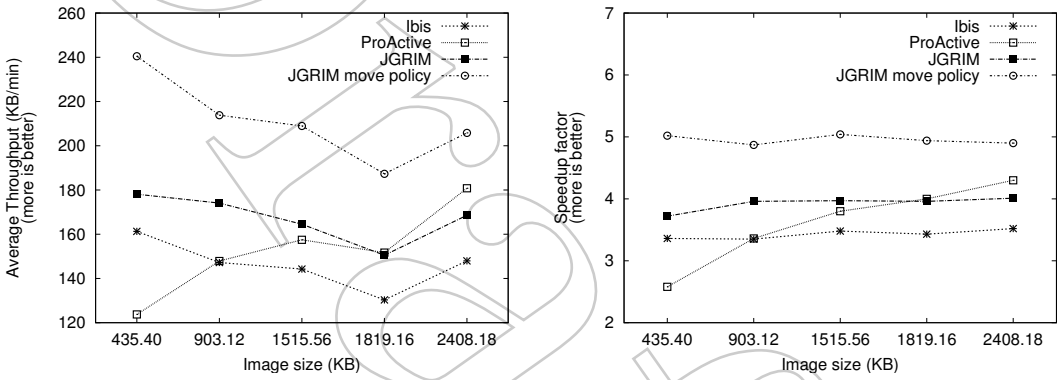
From the figure, it can be observed that JGRIM experienced performance levels similar to the related approaches. Specifically, the resulting JGRIM TET curve is close to the ProActive TET. In addition, the variant of JGRIM using the policy for moving the gridified application brought significant benefits in terms of increased execution performance. Another interesting fact is that coding this policy was very easy. Also,

On the Evaluation of Gridification Effort and Runtime Aspects of JGRIM Applications. (C. Mateos, A. Zunino, M. Campo). Future Generation Computer Systems: The International Journal of Grid Computing: Theory, Methods and Applications. Elsevier Science. ISSN: 0167-739X. Ed.: P. Sloot. Vol. 26, Num. 6, pp 797-819. 2010. DOI: 10.1016/j.future.2010.02.014

as policies are software components that are non-intrusively injected by JGRIM into applications, they do not affect the code of the application logic and can be reused in other applications.

A result that may confuse is that the JGRIM version of the application performed better than the Ibis variant, even when the former used Ibis as the underlying support for application parallelization. The reason of this fact is that the code that is interpreted by the Ibis runtime in either cases is subject to different execution conditions. On one hand, the implementation of the Ibis version comprises the application logic *plus* the code (main method) to run the application, which is called by the Ibis runtime to carry out the handshaking process among Ibis hosts to start and cooperatively execute the application. On the other hand, upon the execution of a self dependency operation in JGRIM (i.e. restoreSubImages), an Ibis object is created and sent by the JGRIM platform to an already deployed Ibis network, which is running a pure Ibis application that is able to execute other Ibis applications. All in all, JGRIM did not performed worse than Ibis, even when JGRIM adds a software layer on top of the execution services of Ibis.

Figures 16 (a) and 16 (b) show the throughput and the speedup achieved by the different variants of the application. Throughput was calculated as the average amount of data processed per time unit, that is, the amount of image data (KB) restored per minute. Speedup represents the time that is necessary to execute the original implementation (estimated by the aforementioned benchmark) over the time required to run the Grid-enabled implementations. As reported, the JGRIM application achieved a throughput similar to that of Ibis and ProActive. Besides, better throughput and speedup was achieved by using mobility.



(a) Throughput (b) Speedup

Figure 16: Throughput (KB/min) and speedup of the restoration applications

Comparison of network traffic. To profile the network resource consumption of the variants of the restoration application, the network traffic (in MB and TCP packets) generated during the test runs was measured. Figure 17 shows the total traffic in MB generated by each variant during the whole experiment, that is, the traffic accumulated throughout the ten runs that were performed to compute TET. Traffic was discriminated into two types: *intra cluster*, which counts the LAN traffic generated to communicate any pair of machines residing in the same cluster, and *extra cluster*, which measures the total network traffic for sending data between two machines belonging to different clusters. Since clusters are connected to each other by Internet links, it is highly desirable to generate little extra cluster traffic because Internet communication is several orders of magnitude more expensive than local communication.

The plain JGRIM variant of the application (without mobility) generated less and more traffic than the Ibis and ProActive implementations, respectively. However, extra cluster communication in the JGRIM version represented the 84% of its total traffic, against a higher value of 89% for ProActive. Naturally, using the mobility policy allowed JGRIM to further reduce the total traffic. Besides, the percentage of extra cluster communication with respect to the total traffic in the policy-based JGRIM application dropped to 57%. In addition, the extra cluster traffic generated by this application was almost 18% lower (30 MB) compared to the extra cluster communication of the ProActive solution. These results evidence an important aspect of JGRIM: policies may be useful not only to achieve higher performance but also to make a better use of Grid network resources.

On the Evaluation of Gridification Effort and Runtime Aspects of JGRIM Applications. (C. Mateos, A. Zunino, M. Campo). Future Generation Computer Systems: The International Journal of Grid Computing: Theory, Methods and Applications. Elsevier Science. ISSN: 0167-739X. Ed.: P. Sloot. Vol. 26, Num. 6, pp 797-819. 2010. DOI: 10.1016/j.future.2010.02.014

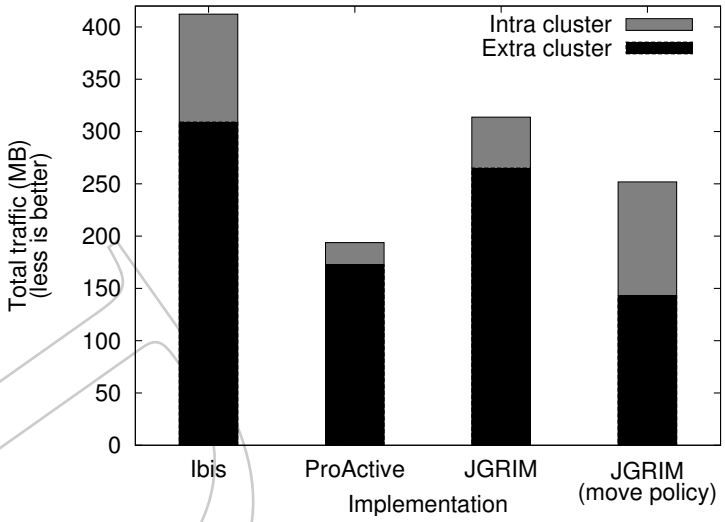


Figure 17: Total traffic (MB) generated during the entire experiment

There are other interesting points that can be observed from the figure. First, the platform that generated the least amount of total network traffic was ProActive. Unlike the rest of the implementations in which nodes randomly try to steal jobs from their peers and thus many messages may be sent before actually get a job to execute, ProActive promotes the use of a job submission model in which the application decides which active object should handle the execution of the next unfinished job. In consequence, less traffic is generated, but more source code have to be provided by the programmer to support this mechanism. Second, similar to the case of TET discussed in the previous subsection, JGRIM accidentally used less network resources than Ibis since both implementations access the core services provided by the Ibis platform under different execution conditions.

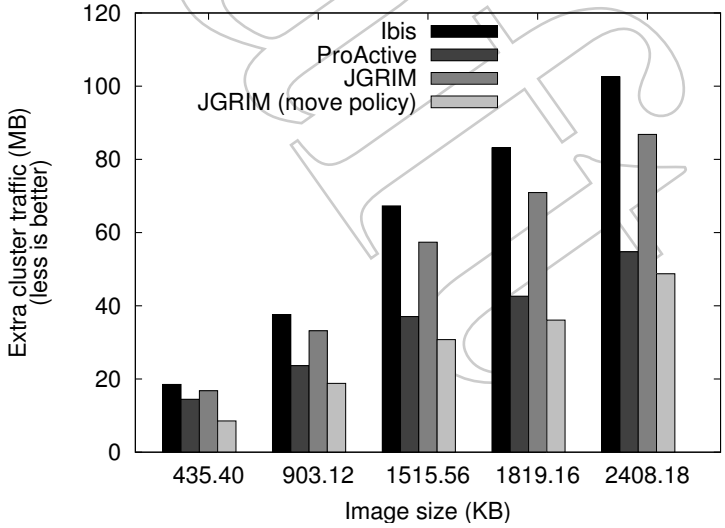


Figure 18: Total traffic (MB) generated during each test battery

As a complement, Figure 18 illustrates the extra cluster network traffic (in MB) generated by the applications during the restoration process of an individual image. Naturally, the total traffic results in higher values as the size of the input image increases. In all cases, the variant of JGRIM with mobility was the most efficient application in terms of network usage. Finally, the percentage of extra cluster packets with

On the Evaluation of Gridification Effort and Runtime Aspects of JGRIM Applications. (C. Mateos, A. Zunino, M. Campo). Future Generation Computer Systems: The International Journal of Grid Computing: Theory, Methods and Applications. Elsevier Science. ISSN: 0167-739X. Ed.: P. Sloot. Vol. 26, Num. 6, pp 797-819. 2010. DOI: 10.1016/j.future.2010.02.014

respect to the total generated packets was 52 (Ibis), 89 (ProActive), 72 (JGRIM) and 57 (JGRIM with policy). In consequence, the non-mobile and mobile JGRIM implementations of the application experienced network performance levels –with respect to TCP packets– similar to that of ProActive and Ibis, respectively. The amount of packets used to carry out local communication can be ignored, as they are subject to very small latency values compared to extra cluster communication. Finally, in both JGRIM implementations, a considerable percentage of extra cluster packets corresponded to communication performed by the underlying Ibis execution services.

5. Conclusions

In this paper, we have presented JGRIM, a new approach to simplify the gridification of conventional Java applications. The utmost goal of JGRIM is to isolate developers as much as possible from the complexities of the Grid and its services. We showed the advantages of JGRIM through comparisons with Ibis and ProActive, two Java-based platforms for Grid application development that materialize alternative approaches to gridification. Ibis, ProActive and JGRIM were used for gridifying two applications, namely the k-NN algorithm and an application for enhancement of panoramic images. We then performed an analysis on the Grid-enabled codes via a novel formula that estimates gridification effort. Moreover, experiments related to execution time and network usage of these applications were conducted on an Internet-connected Grid.

All in all, the reported experiments showed that JGRIM simplifies gridification and better preserves the application logic. Consequently, the Grid-aware code is easier to maintain, test, and port to different Grid platforms. In addition, experiments suggest that, even when JGRIM may cause gridified applications to consume more Grid resources than its related approaches, this do not directly translate into an irremediable problem, since policies can be used to easily and non-intrusively improve resource usage and to allow JGRIM applications to perform in a very competitive way. Furthermore, although the evaluation conceived Ibis and ProActive as competitors of JGRIM, these platforms are in some respect complementary to our work. Essentially, JGRIM promotes separation of concerns between application logic and Grid behavior, being this latter Grid services provided by existing applications and *platforms*. In this way, JGRIM provides an alternative method for gridifying applications while it does not "reinvent the wheel" by providing Grid execution services for these applications. In fact, JGRIM is currently able to leverage the services of Ibis. Besides, efforts to integrate JGRIM with ProActive and Condor are underway.

It is worth emphasizing that the goal of the experiments was not only to evaluate the performance of JGRIM but also to quantify the effort necessary to gridify applications. The comparison of gridification effort among the employed approaches was achieved through a novel metric (GE) that considers the dimensions of the gridification problem first identified in Mateos et al. [12]. On the other hand, the rest of the experiments were performed based on a Grid setting with extremely heterogeneous hardware capabilities as well as different and very dissimilar public Internet links, therefore ensuring environment heterogeneity and complexity. Moreover, recent experiments Mateos et al. [58, 59] performed in the context of the BYG project Mateos et al. [60] with seven CPU-intensive applications and a larger heterogeneous simulated wide-area Grid, suggest that JGRIM metaservices add an acceptable overhead that does not depend on the size of the setting. BYG is essentially a JGRIM-compliant middleware for just-in-time gridification of Java bytecode. In addition, to measure the impact of metaservices in the resulting performance, we have shown that GMAC Gotthelf et al. [61] –a P2P protocol of our own currently used by JGRIM to manage host communication– is scalable enough to support Grids of thousands of nodes while introducing a small overhead in terms of network resource usage Gotthelf et al. [61]. Despite these encouraging results, we will further experiment with JGRIM to provide more evidence on its performance and thus delineate future optimizations. For example, we are working on the gridification of several scientific Java-based applications (e.g. sequence alignment) on a large real (not simulated) wide-area Grid. The infrastructure is a result of a country-wide Grid initiative of the Argentinian government that will connect academic clusters across different provinces of Argentina, which was officially initiated on October 2009[10], but it is not yet available for experimentation purposes.

[10] *http://indico.cern.ch/conferenceProgram.py?confId=66398*

On the Evaluation of Gridification Effort and Runtime Aspects of JGRIM Applications. (C. Mateos, A. Zunino, M. Campo). Future Generation Computer Systems: The International Journal of Grid Computing: Theory, Methods and Applications. Elsevier Science. ISSN: 0167-739X. Ed.: P. Sloot. Vol. 26, Num. 6, pp 797-819. 2010. DOI: 10.1016/j.future.2010.02.014

At present, JGRIM is being extended in several directions. We are working on tools to make JGRIM easier to adopt and use. We have developed a prototype implementation of an Eclipse plug-in that lets developers to gridify their applications by graphically identifying hot-spots, configuring policies, etc. Eventually, this plug-in will also offer proper support for deploying and monitoring the execution of applications. It is expected that the plug-in will also let developers to inspect the execution state of gridified applications for debugging purposes. In summary, the goal of this line of research is to supply programmers with a full-fledged IDE for developing JGRIM applications.

Also, we are exploring the viability of materializing JGRIM in other programming languages, such as C++, Python, Ruby or Perl. The main motivation behind this is that many of these languages, specially C++, are being extensively used for programming Grid and parallel applications. In this way, developers will be able to take advantage of JGRIM by using the programming language of their choice. Note that materializing JGRIM concepts to a particular language will require to study whether the language supports core features such as application migration, dependency injection and Web Service invocation capabilities. Fortunately, a number of DI frameworks for C++, Python, Ruby and Perl already exist. Moreover, APIs for invoking Web Services and libraries implementing process migration techniques for some of these languages also exist.

Finally, another issue that will be addressed is the use of scalable mechanisms for service discovery. Currently, JGRIM employs a centralized discovery scheme developed on top of UDDI registries. However, tomorrow's Grids will offer thousands if not millions of services, and so will be the number of potential clients for these services. In this context, scalability will be crucial, therefore rendering solutions for service discovery based on centralized mechanisms totally inappropriate. Precisely, a good alternative to centralized service discovery are P2P technologies [62]. In this sense, a line toward this end is the extension of GMAC with service discovery capabilities.

References

[1] I. Foster, C. Kesselman, S. Tuecke, The anatomy of the Grid: Enabling scalable virtual organization, International Journal of High Performance Computing Applications 15 (3) (2001) 200–222.

[2] I. Foster, The Grid: Computing without bounds, Scientific American 288 (4) (2003) 78–85.

[3] M. Chetty, R. Buyya, Weaving computational Grids: How analogous are they with electrical Grids?, Computing in Science and Engineering 4 (4) (2002) 61–71.

[4] A. Bazinet, D. Myers, J. Fuetsch, M. Cummings, Grid Services Base Library: A high-level, procedural application programming interface for writing Globus-based Grid services, Future Generation Computer Systems 23 (3) (2007) 517–522.

[5] G. von Laszewski, J. Gawor, P. Lane, N. Rehn, M. Russell, Features of the Java commodity Grid kit, Concurrency and Computation: Practice and Experience, Special Issue on Grid Computing Environments 14 (13-15) (2003) 1045–1055.

[6] A. Paventhan, K. Takeda, S. Cox, D. Nicole, MyCoG.NET: A multi-language CoG toolkit, Concurrency and Computation: Practice and Experience, Special Issue on Middleware for Grid Computing: 'A Possible Future' 19 (14) (2006) 1885–1900.

[7] 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: Towards generic and easy application programming interfaces for the Grid, Proceedings of the IEEE 93 (3) (2005) 534–550.

[8] T. Goodale, S. Jha, H. Kaiser, T. Kielmann, P. Kleijer, G. von Laszewski, C. Lee, A. Merzky, H. Rajic, J. Shalf, SAGA: A Simple API for Grid Applications - high-level application programming on the Grid, Computational Methods in Science and Technology 12 (1) (2006) 7–20.

[9] J.-P. Goux, S. Kulkarni, M. Yoder, J. Linderoth, Master-Worker: An enabling framework for applications on the Computational Grid, Cluster Computing 4 (1) (2001) 63–70.

[10] F. Berman, R. Wolski, H. Casanova, W. Cirne, H. Dail, M. Faerman, S. Figueira, J. Hayes, G. Obertelli, J. Schopf, G. Shao, S. Smallen, N. Spring, A. Su, D. Zagorodnov, Adaptive computing on the Grid using AppLeS, IEEE Transactions on Parallel Distributed Systems 14 (4) (2003) 369–382.

[11] J. Ferreira, J. Sobral, A. Proenca, JaSkel: A Java skeleton-based framework for structured cluster and Grid computing, in: 6th IEEE International Symposium on Cluster Computing and the Grid (CCGrid '06), Singapore, IEEE Computer Society, Washington, DC, USA, 2006, pp. 301–304.

[12] C. Mateos, A. Zunino, M. Campo, A survey on approaches to gridification, Software: Practice and Experience 38 (5) (2008) 523–556.

[13] C. Mateos, A. Zunino, M. Campo, JGRIM: An approach for easy gridification of applications, Future Generation Computer Systems 24 (2) (2008) 99–118.

[14] R. van Nieuwpoort, J. Maassen, G. Wrzesinska, R. Hofman, C. Jacobs, T. Kielmann, H. Bal, Ibis: A flexible and efficient Java based grid programming environment, Concurrency and Computation: Practice and Experience 17 (7-8) (2005) 1079–1107.

[15] J. Dongarra, D. Walker, MPI: A standard Message Passing Interface, Supercomputer 12 (1) (1996) 56–68.

[16] T. Downing, Java RMI: Remote Method Invocation, IDG Books Worldwide, Foster City, CA, USA, 1998.

[17] R. van Nieuwpoort, J. Maassen, T. Kielmann, H. Bal, Satin: Simple and efficient Java-based Grid programming, Scalable Computing: Practice and Experience 6 (3) (2005) 19–32.

On the Evaluation of Gridification Effort and Runtime Aspects of JGRIM Applications. (C. Mateos, A. Zunino, M. Campo). Future Generation Computer Systems: The International Journal of Grid Computing: Theory, Methods and Applications. Elsevier Science. ISSN: 0167-739X. Ed.: P. Sloot. Vol. 26, Num. 6, pp 797-819. 2010. DOI: 10.1016/j.future.2010.02.014

[18] G. Wrzesinska, R. van Nieuwport, J. Maassen, T. Kielmann, H. Bal, Fault-tolerant scheduling of fine-grained tasks in grid environments, International Journal of High Performance Computing Applications 20 (1) (2006) 103–114.
[19] W3C Consortium, Web Services Description Language (WSDL) Version 2.0 Part 1: Core Language, W3C Candidate Recommendation, http://www.w3.org/TR/wsdl20 (Jun. 2007).
[20] OASIS Consortium, UDDI version 3.0.2, UDDI Spec Technical Committee Draft, http://uddi.org/pubs/uddi_v3.htm (Oct. 2004).
[21] H. Stockinger, Defining the Grid: A snapshot on the current view, Journal of Supercomputing 42 (1) (2007) 3–17.
[22] A. Munawar, M. Wahib, M. Munetomo, K. Akama, The design, usage, and performance of GridUFO: A Grid based Unified Framework for Optimization, Future Generation Computer Systems 26 (4) (2010) 633–644.
[23] L. Baduel, F. Baude, D. Caromel, A. Contes, F. Huet, M. Morel, R. Quilici, Grid Computing: Software Environments and Tools, Springer, Berlin, Heidelberg, and New York, 2006, Ch. Programming, Composing, Deploying on the Grid, pp. 205–229.
[24] D. Caromel, A. di Costanzo, C. Delbé, Peer-to-peer and fault-tolerance: Towards deployment-based technical services, Future Generation Computer Systems 23 (7) (2007) 879–887.
[25] A. Jugravu, T. Fahringer, JavaSymphony, a programming model for the Grid, Future Generation Computer Systems 21 (1) (2005) 239–247.
[26] D. Gannon, S. Krishnan, L. Fang, G. Kandaswamy, Y. Simmhan, A. Slominski, On building parallel and Grid applications: Component technology and distributed services, Cluster Computing 8 (4) (2005) 271–277.
[27] I. Foster, Globus Toolkit version 4: Software for service-oriented systems, in: Network and Parallel Computing - IFIP International Conference (NPC '05), Beijing, China, Vol. 3779, Springer, 2005, pp. 2–13.
[28] T. Delaittre, T. Kiss, A. Goyeneche, G. Terstyanszky, S. Winter, P. Kacsuk, GEMLCA: Running legacy code applications as Grid services, Journal of Grid Computing 3 (1-2) (2005) 75–90.
[29] J. Kommineni, D. Abramson, GriddLeS enhancements and building virtual applications for the Grid with legacy components, in: P. M. A. Sloot, A. Hoekstra, T. Priol, A. Reinefeld, M. Bubak (Eds.), Advances in Grid Computing - European Grid Conference (EGC '05), Amsterdam, The Netherlands, Vol. 3470 of Lecture Notes in Computer Science, Springer, 2005, pp. 961–971.
[30] V. Sanjeepan, A. Matsunaga, L. Zhu, H. Lam, J. Fortes, A service-oriented, scalable approach to Grid-enabling of legacy scientific applications, in: 3th IEEE International Conference on Web Services (ICWS'05), Orlando, Florida, USA, IEEE Computer Society, Los Alamitos, CA, USA, 2005, pp. 553–560.
[31] Q. Ho, T. Hung, W. Jie, H. Chan, E. Sindhu, S. Ganesan, T. Zang, X. Li, GRASG - a framework for 'gridifying' and running applications on service-oriented Grids, in: 6th IEEE International Symposium on Cluster Computing and the Grid (CCGrid '06), Singapore, IEEE Computer Society, Washington, DC, USA, 2006, pp. 305–312.
[32] M. B. Bartosz Baliś, M. Wegiel, LGF: A flexible framework for exposing legacy codes as services, Future Generation Computer Systems 24 (7) (2008) 711–719.
[33] D. Thain, T. Tannenbaum, M. Livny, Condor and the grid, in: F. Berman, G. Fox, A. Hey (Eds.), Grid Computing: Making the Global Infrastructure a Reality, John Wiley & Sons, Inc., New York, NY, USA, 2003, pp. 299–335.
[34] K. Czajkowski, C. Kesselman, S. Fitzgerald, I. Foster, Grid information services for distributed resource sharing, in: 10th IEEE International Symposium on High Performance Distributed Computing (HPDC '01), San Francisco, CA, USA, IEEE Computer Society, Washington, DC, USA, 2001, pp. 181–194.
[35] C. Mateos, A. Zunino, M. Campo, Extending MoviLog for supporting Web Services, Computer Languages, Systems & Structures 33 (1) (2007) 11–31.
[36] R. Johnson, J2EE development frameworks, Computer 38 (1) (2005) 107–110.
[37] M. Atkinson, D. DeRoure, A. Dunlop, G. Fox, P. Henderson, T. Hey, N. Paton, S. Newhouse, S. Parastatidis, A. Trefethen, P. Watson, J. Webber, Web Service Grids: An evolutionary approach, Concurrency and Computation: Practice and Experience 17 (2-4) (2005) 377–389.
[38] OGSA-WG, Defining the Grid: A roadmap for OGSA standards, http://www.ogf.org/documents/GFD.53.pdf (Sep. 2005).
[39] OASIS Consortium, Web Services Resource Framework (WSRF) - Primer v1.2. Committee Draft 02, http://docs.oasis-open.org/wsrf/wsrf-primer-1.2-primer-cd-02.pdf (May 2006).
[40] C. Aiftimiei, P. Andreetto, S. Bertocco, S. Dalla Fina, A. Dorigo, E. Frizziero, A. Gianelle, M. Marzolla, M. Mazzucato, M. Sgaravatto, S. Traldi, L. Zangrando, Design and implementation of the gLite CREAM job management service, Future Generation Computer Systems 26 (4) (2010) 654–667.
[41] E. Lee, B. Lee, W. Shin, C. Wu, A reengineering process for migrating from an object-oriented legacy system to a component-based system, in: Design and Assessment of Trustworthy Software-Based Systems - 27th International Conference on Computer Software and Applications (COMPSAC '03), Dallas, Texas, USA, IEEE Computer Society, Washington, DC, USA, 2003, pp. 336–341.
[42] S. D. Kim, S. H. Chang, A systematic method to identify software components, in: 11th Asia-Pacific Software Engineering Conference (APSEC '04), Busan, Korea, IEEE Computer Society, Washington, DC, USA, 2004, pp. 538–545.
[43] S. Li, L. Tahvildari, JComp: A reuse-driven componentization framework for Java applications, in: 14th IEEE International Conference on Program Comprehension (ICPC '06), Athens, Greece, IEEE Computer Society, Los Alamitos, CA, USA, 2006, pp. 264–267.
[44] D. Thain, M. Livny, Error scope on a computational Grid: Theory and practice, in: 11th IEEE Symposium on High Performance Distributed Computing (HPDC-11 '02), Edinburgh, Scotland, IEEE Computer Society, Washington, DC, USA, 2002, pp. 199–208.
[45] B. Allcock, J. Bester, J. Bresnahan, A. Chervenak, I. Foster, C. Kesselman, S. Meder, V. Nefedova, D. Quesnel, S. Tuecke, Data management and transfer in high-performance computational Grid environments, Parallel Computing 28 (5) (2002) 749–771.
[46] B. Dasarathy (Ed.), Nearest Neighbor (Nn) Norms: Nn Pattern Classification Techniques, IEEE Computer Society, Los Alamitos, CA, USA, 1991.
[47] M. Feilner, OpenVPN: Building and Integrating Virtual Private Networks, Packt Publishing Ltd., Birmingham, UK, 2006.

On the Evaluation of Gridification Effort and Runtime Aspects of JGRIM Applications. (C. Mateos, A. Zunino, M. Campo). Future Generation Computer Systems: The International Journal of Grid Computing: Theory, Methods and Applications. Elsevier Science. ISSN: 0167-739X. Ed.: P. Sloot. Vol. 26, Num. 6, pp 797-819. 2010. DOI: 10.1016/j.future.2010.02.014

[48] S. Chidamber, C. Kemerer, A metrics suite for object oriented design, IEEE Transactions on Software Engineering 20 (6) (1994) 476–493.
[49] R. Martin, Object-oriented design quality metrics: An analysis of dependencies, Report on Object Analysis and Design 2 (3).
[50] W. Li, S. Henry, Object-oriented metrics that predict maintainability, Journal of Systems Software 23 (2) (1993) 111–122.
[51] G. Wrzesinska, J. Maassen, K. Verstoep, H. Bal, Satin++: Divide-and-share on the Grid, in: 2nd IEEE International Conference on e-Science and Grid Computing (E-SCIENCE '06), Amsterdam, Netherlands, IEEE Computer Society, Washington, DC, USA, 2006, p. 61.
[52] W3C Consortium, SOAP version 1.2 part 0: Primer (second edition), W3C Recommendation, http://www.w3.org/TR/soap12-part0 (Apr. 2007).
[53] V. Freeh, A comparison of implicit and explicit parallel programming, Journal of Parallel and Distributed Computing 34 (1) (1996) 50–65.
[54] D. Eager, J. Zahorjan, E. Lozowska, Speedup versus efficiency in parallel systems, IEEE Transactions on Computers 38 (3) (1989) 408–423.
[55] D. Tschumperlé, R. Deriche, Vector-valued image regularization with PDE's: A common framework for different applications, in: IEEE Conference on Computer Vision and Pattern Recognition (CVPR '03), Madison, WI, USA, Vol. 1, IEEE Computer Society, Los Alamitos, CA, USA, 2003, pp. 651–656.
[56] D. Caromel, C. Delbé, A. di Costanzo, M. Leyton, ProActive: An integrated platform for programming and running applications on Grids and P2P systems, Computational Methods in Science and Technology 12 (1) (2006) 69–77.
[57] A. Milanés, N. Rodriguez, B. Schulze, State of the art in heterogeneous strong migration of computations, Concurrency and Computation: Practice and Experience 20 (13) (2008) 1485–1508.
[58] C. Mateos, A. Zunino, M. Campo, A tool for automatic parallelization of CPU-intensive Java applications on distributed environments, Mecánica Computacional XXCIII (2009) 253–272.
[59] C. Mateos, A. Zunino, M. Campo, N. Gómez, Just-in-time gridification of compiled Java applications, in: II Argentine Symposium on High Performance Computing (HPC2009) - 38th Jornadas Argentinas de Informática e Investigación Operativa (JAIIO 2009), Mar del Plata, Buenos Aires, Argentina, SADIO, Buenos Aires, Argentina, 2009, pp. 62–77.
[60] C. Mateos, A. Zunino, M. Campo, R. Trachsel, Parallel Programming, Models and Applications in Grid and P2P Systems, Advances in Parallel Computing, IOS Press, Amsterdam, The Netherlands, 2009, Ch. BYG: An Approach to Just-in-Time Gridification of Conventional Java Applications, pp. 232–260.
[61] P. Gotthelf, A. Zunino, C. Mateos, M. Campo, GMAC: An overlay multicast network for mobile agent platforms, Journal of Parallel and Distributed Computing 68 (8) (2008) 1081–1096.
[62] J. Garofalakis, Y. Panagis, E. Sakkopoulos, A. Tsakalidis, Contemporary Web Service discovery mechanisms, Journal of Web Engineering 5 (3) (2006) 265–290.

Acknowledgments

We thank the anonymous reviewers for their comments and suggestions to improve the paper. We acknowledge the financial support provided by ANPCyT through grants PAE-PICT 2007-02311 and PAE-PICT 2007-02312.

About the authors

Cristian Mateos received a Ph.D. degre in Computer Science from UNICEN, Tandil, Argentina, in 2008. He is a Teacher Assistant at the Computer Science Department of UNICEN. His thesis was on solutions to ease Grid application development and tuning through dependency injection and policies.

Alejandro Zunino received a Ph.D. degree in Computer Science from UNICEN in 2003. He is an Assistant Professor at the Computer Science Department of UNICEN and a research fellow of the CONICET. He has published over 40 papers in journals and conferences.

Marcelo Campo received a Ph.D. degree in Computer Science from UFRGS, Porto Alegre, Brazil. He is an Associate Professor at the Computer Science Department and Head of the ISISTAN. He is also a research fellow of the CONICET. He has over 70 papers published in conferences and journals about software engineering topics.