Future Generation Computer Systems 26 (2010) 797–819

Contents lists available at ScienceDirect

Future Generation Computer Systems

journal homepage: www.elsevier.com/locate/fgcs

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
Consejo Nacional de Investigaciones Científicas y Técnicas (CONICET), Argentina

ARTICLE INFO

Article history:
Received 27 March 2008
Received in revised form
16 February 2010
Accepted 25 February 2010
Available online 6 March 2010

Keywords:
Grid computing
Gridification
Grid services
JGRIM
Dependency injection

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 by Grid-aware code, as a result of which the 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.

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 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

* Corresponding address: ISISTAN Research Institute, UNICEN University, Campus Universitario, Tandil (B7001BBO), Buenos Aires, Argentina. Tel.: +54 2293 439682x35; fax: +54 2293 439683.
E-mail address: cmateos2006@gmail.com (C. Mateos).

0167-739X/$ – see front matter
doi:10.1016/j.future.2010.02.014

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. Here 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 a 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 [21]. These technologies thus supply the basis for more interoperable Grids and underlie several of the current Grid initiatives [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 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. [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 [28–32]. 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 [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 [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

[1] JavaBeans http://java.sun.com/javase/technologies/desktop/javabeans/docs/spec.html.

[2] Enterprise JavaBeans http://java.sun.com/products/ejb.