Advances in Engineering Software 42 (2011) 172–186

Contents lists available at ScienceDirect

Advances in Engineering Software

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

A software tool for semi-automatic gridification of resource-intensive Java bytecodes and its application to ray tracing and sequence alignment

Cristian Mateos [a,c,*], Alejandro Zunino [a,c], Matías Hirsch [b], Mariano Fernández [b], Marcelo Campo [a,c]

[a] ISISTAN Research Institute – Campus Universitario, Tandil, Buenos Aires, Argentina
[b] UNICEN University – Tandil, Buenos Aires, Argentina
[c] Consejo Nacional de Investigaciones Científicas y Técnicas – Ciudad Autónoma de Buenos Aires, Buenos Aires, Argentina

ARTICLE INFO

Article history:
Received 9 January 2011
Accepted 15 February 2011

Keywords:
Computational grids
Gridification
Resource-intensive applications
Automatic parallelism
Parallelization heuristics
Java bytecode

ABSTRACT

Computational Grids deliver the necessary computational infrastructure to perform resource-intensive computations such as the ones that solve the problems scientists are facing today. Exploiting Computational Grids comes at the expense of explicitly adapting the ordinary software implementing scientific problems to take advantage of Grid resources, which unavoidably requires knowledge on Grid programming. The recent notion of "gridifying" ordinary applications, which is based on semi-automatically deriving a Grid-aware version from the compiled code of a sequential application, promises users to be relieved from the requirement of manual usage of Grid APIs within their source codes. In this paper, we describe a novel gridification tool that allows users to easily parallelize Java applications on Grids. Extensive experiments with two real-world applications – ray tracing and sequence alignment – suggest that our approach provides a convenient balance between ease of gridification and Grid resource exploitation compared to manually using Grid APIs for gridifying ordinary applications.

1. Introduction

Computational Grids are distributed heterogeneous clusters that allow scientists to build applications that demand by nature a huge amount of computational resources such as CPU cycles and memory [14]. Examples of such applications include aerodynamic design, weather prediction, catastrophe simulation, financial modeling, drug discovery, amongst others. The sad part of the story is that taking advantage of such computational infrastructures requires significant development effort and knowledge on distributed as well as parallel programming. In other words, there is a very high coupling between the tasks of writing the sequential implementation of the algorithm that represent a simulation and obtaining its Grid-enabled version. As a consequence, at development time, a user must take into account the functional aspects of his application (what the application does) as well as many details of the underlying Grid execution infrastructure (how the application executes). Clearly, the second requirement cannot be easily accomplished by scientists and practitioners not proficient in Grid programming.

The traditional approach to cope with the problem of easily exploiting Grids is based on supplying users with programming APIs such as MPI [44] and PVM [44], which provide standard and simple interfaces to Grids through the provision of primitives to execute parts of an application in a distributed and coordinated way. To this end, a user must in principle indicate which parts of its application can benefit from being parallelized by inserting in the sequential code that implements his application appropriate calls to such primitives. Interestingly, APIs like MPI and PVM mitigate the complexity inherent to writing Grid applications as they encapsulate common distributed and parallel patterns behind an intuitive API. However, such APIs still require users to have a solid knowledge in parallel and distributed programming, which prevents inexperienced users (e.g. scientists or engineers) from effectively taking advantage of Grid technologies [53].

More recently, the notion of "gridifying" sequential applications [35] has appeared as a fresh approach for rapidly developing and seamlessly running applications on Computational Grids. Basically, gridification tools seek to avoid the manual usage of APIs for distributed and parallel programming within the source code of user applications and otherwise automatically derive the Grid counterparts from the (sequential) compiled code of these applications. However, materializing the concept is indeed challenging, as it is intuitively very difficult to automatically transform a sequential application to run on a Grid and still deeply exploit parallelism in the application to boost its performance.

In this paper, we describe a novel Java-based gridification tool called BYG (BYtecode Gridifier), which operates by using some novel techniques for modifying and parallelizing bytecodes – i.e.

* Corresponding author at: ISISTAN Research Institute, Argentina. Tel.: +54 2293 439682x35; fax: +54 2293 439681.
E-mail address: cmateos@conicet.gov.ar (C. Mateos).

0965-9978/$ - see front matter © 2011 Elsevier Ltd. All rights reserved.
doi:10.1016/j.advengsoft.2011.02.003

the binary code flavor generated by the compiler of the Java language – to produce efficient Grid applications. Basically, the idea is to support users who would like to quickly parallelize and run their sequential codes on a Grid without dealing with typically complex Grid programming and infrastructure details. Furthermore, the current materialization of BYG targets Java applications developed under the divide and conquer model, a well-known technique for algorithm design by which a problem is solved by systematically dividing it into several subproblems until trivial subproblems are obtained, which are solved directly. Basically, upon executing an ordinary application, its bytecode is modified so that it is able to execute such subproblems in parallel using the nodes of a Grid.

Preliminary experiments with our tool in a small LAN and resource-intensive benchmark applications showed the feasibility of the approach [37]. Here, we evaluate BYG by gridifying and running two resource-intensive and real-world applications, namely ray tracing and sequence alignment, on a wide-area Computational Grid. The former application is a popular rendering technique that outputs a picture using an abstract description of a 3D scene, while the latter is an algorithm for comparing gene sequences, a well-known problem in bioinformatics. Furthermore, we derived variants of these applications by manually parallelizing them via the GridGain [21] and Satin [54] Grid libraries, which are designed for parallelizing and efficiently executing applications on both clusters and Grids. The comparisons suggest that BYG offers a convenient alternative to the problem of easy gridification of sequential applications, while delivers acceptable performance and fair resource usage compared to manual parallelism. On the other hand, given the ever increasing popularity of the Java language for distributed programming, which is mostly explained by its platform-neutral bytecode and its very good performance in large-scale distributed environments compared to traditional languages [47], and the simplicity and versatility of the divide and conquer model, we believe that BYG is an attractive alternative for painlessly gridifying a broad range of resource-intensive applications.

The rest of the paper is organized as follows. Section 2 discusses the most relevant related works. Section 3 overviews BYG and explains how our approach improves over them. For the most part, the Section describes the use of BYG in the context of a specific Grid scheduler library, for which the current version of BYG provides integration. Section 4 reports the abovementioned experimental evaluation. Section 5 concludes the paper and discusses prospective future works.

2. Related work

The two common approaches that researchers have been followed to address the problem of simplifying the development of high-performance scientific applications are based on either providing domain-specific solutions or general–purpose tools. The first approach aims at providing APIs and runtime supports for taking advantage of widely-employed scientific libraries from within applications. Alternatively, the second approach allows users to implement applications while not necessarily relying on specific scientific libraries. Both approaches have their pros and cons, as detailed below.

Among the efforts that follow the first approach is the work by Baitsch and his colleagues [6], which propose a Java toolkit for writing numerical intensive applications. The toolkit builds on the efficiency of numerical Fortran libraries such as BLAS, LAPACK and NAG by providing Java wrappers that directly access the corresponding native libraries via the Java-to-C interface. In addition, the toolkit provides a Java-based library that comprise classes for common vector, matrix and linear algebra operations. Similarly, f2j [45] is a Fortran-to-Java translator specially designed to obtain the Java counterpart of the Fortran code of the BLAS and LAPACK libraries (this latter is codenamed JLAPACK [11]). Moreover, the jLab environment [43] offers a scripting language similar to Matlab and Scilab for programming applications that are executed by an interpreter implemented in Java. This environment supports the basic programming constructs of Matlab (e.g. operators for manipulating matrixes) and is embedded in a graphical development environment. Furthermore, the work by Eyheramendy [13] proposes a Java-based library for building Computational Fluid Dynamics applications. In its current shape, the framework supports different finite elements formulations for basics mechanical problems, and some of them can be parallelized by using multithreaded programming.

Indeed, the idea of providing domain-specific tools is not only circumscribed to Java, as evidenced by similar supports for other programming languages. An example is PyScaLAPACK [12], a Python interface to ScaLAPACK [40]. ScaLAPACK is a subset of the LAPACK linear algebra routines but adapted for cluster computing by using the MPI [44] or the PVM [44] parallel libraries. Moreover, the work by Mackie [32] proposes a finite element distributed solver written in the .NET platform. However, the two negative characteristics of the efforts following the approach discussed so far is that they restrict the kind of applications that can be written and, except for few cases, they are not capable of exploiting clusters and Grid infrastructures. Among the tools that do exploit distributed environments, some works that deserve mention are the Alya system [7], which provides several kernels for programming and executing various types of Computational Mechanics applications in parallel on large-scale clusters, and GMarte [2], a middleware for programmatically building and running task-based applications on Computational Grids, which has been recently applied to 3D analysis of large dimension buildings [3].

Precisely, MPI and PVM are the oldest standards for building general-purpose parallel applications. When using these libraries, applications are parallelized by decomposing them into a number of distributed components that communicate via message exchange. Several Java bindings for MPI (e.g. mpiJava [25], MPJ Express [46]), PVM (e.g. jPVM [51]) or both (JCluster [55]) exists. However, MPI and PVM have also received much criticism [31] since they are basically low-level parallelization tools that require solid knowledge on both parallel programming and distributed deployment from users. In response, there are some Java tools that attempt to address these problems by raising the level of abstraction of the API exposed to users and relieving them as much as possible from performing parallelization and deployment tasks.

Particularly, ProActive [5] is a Java platform for parallel distributed computing that provides *technical services*, a flexible support to address non-functional Grid concerns (e.g. load balancing and fault tolerance) by plugging configuration external to applications at deployment time. Moreover, ProActive features integration with a wide variety of Grid schedulers, and supports execution of Scilab scripts on dedicated clusters. JavaSymphony [27] is a performance-oriented platform featuring a semi-automatic execution model that automatically deals with parallelism and load balancing of Grid applications, and at the same time allows programmers to control such features via API calls. Unfortunately, using these API-inspired parallelization tools unavoidably requires to learn and manually use their associated APIs within the source code of the (sequential) user application, which compromises usability since these tasks are difficult to achieve for an average programmer.

In consequence, some tools aimed at further simplifying the complexity of the exposed parallel library API and thus improving usability have been proposed, such as VCluster [56] and DG-ADAJ [30]. VCluster supports execution of thread-based Java applications on multicore clusters by relying on a thread migration technique