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).

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 that achieves efficient dynamic load balancing of threads across the nodes of a cluster. Similarly, DG-ADAJ provides a mechanism for transparent execution of multithreaded Java applications on desktop PC Grids. DG-ADAJ automatically derives graphs from the bytecode of a Java application by using representative sets of input data. The graphs account for data and control dependencies within the application. Then, a scheduling heuristic is applied to place mutually exclusive execution paths extracted from the graphs among the nodes of a cluster. The weak point of VCluster and DG-ADAJ is that they promote threads as the base parallel programming model, which makes programming, testing and debugging of applications rather difficult due to the non-deterministic nature of thread execution [31].

In this sense, the Satin framework [54] avoids the explicit usage of threads while achieves semi-automatic parallelization and distribution of subcomputations by targeting recursive (divide and conquer) applications and modifying the compiled code of an application to handle the execution of parallel tasks on a Grid. The user is responsible for indicating in the application code the points in which a fork (i.e. calls to recursive methods) or a join (i.e. to wait for child computations) should take place. A similar framework for .NET applications is Volta [34], which recompiles executables on the basis of declarative developer annotations in order to insert remoting and synchronization primitives to transform applications into their distributed form. Still, tools like Satin and Volta require some modifications to the source code of a user application to insert parallel-specific API code prior to actually Grid-enabling their compiled counterpart. The same problem is also exhibited by some parallelization tools for Java (e.g. PAL [10], GridGain [9]) that use annotations in the source code of sequential applications. Annotated codes are then preprocessed to generate Grid-enabled valid Java code.

Finally, another line of approaches to gridification that effectively minimize any form of code modification in the input sequential application are those promoting separation of concerns between the functional aspects of the application (i.e. its pure behavior) and the Grid-specific behavior [23,33,19]. This is commonly achieved via aspect-oriented programming (AOP) [28] techniques, whereby a sequential code is attached one or more "aspects" that encapsulate how the different portions of this code are executed in parallel within a Grid. The weak point of these approaches is that they unnecessarily impose a specific development paradigm (i.e. AOP) which most developers from the scientific community are not familiar with.

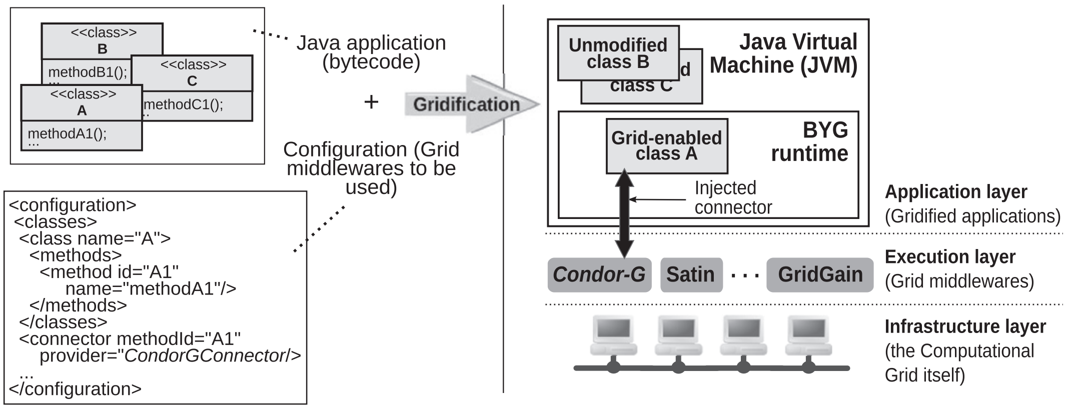
Our tool differs from the abovementioned works in several respects. Firstly, BYG is not targeted at a specific application domain, but can be used to parallelize codes coming from many scientific areas. Secondly, BYG is based on the pervasive as well as intuitive divide and conquer programming model, an algorithmic abstraction that is present in many real-world problems. Thirdly, BYG allows novice users to semi-automatically introduce parallelism into the compiled version of applications, which avoids the requirement of learning parallel programming APIs and altering their codes. In summary, the contribution behind BYG is a software tool to easily and non-intrusively parallelize a wide range of resource-intensive scientific codes so as to take advantage of Computational Grids.

3. The BYG (BYtecode Gridifier) approach

BYG (BYtecode Gridifier) is a new general-purpose gridification tool that allows developers gridifying their applications with minimal effort. To this end, BYG removes the need to explicitly alter sequential application codes, and avoids imposing complex parallel programming models not suitable for users with limited knowledge on Grid programming. In addition, BYG does not seek to provide yet another runtime system for supporting distributed and parallel application execution, but aims at leveraging the schedulers of existing Grid platforms through the use of *connectors*. A connector implements the bridge to access the execution services of a specific Grid platform. Connectors are non-invasively injected into the input sequential application to delegate the execution of certain parts of the application to a Grid platform. The mapping of which parts of the application are Grid-enabled is specified by means of user-supplied configuration external to the sequential code being gridified.

Fig. 1 depicts an overview of BYG. Conceptually, our approach takes as input the bytecode or executable code of an ordinary Java application, and dynamically transforms their classes to run some methods on different Grid middlewares. The developer must indicate through a configuration file which Java methods should be run on a Grid and which Grid middlewares should be used. Then, BYG processes the configuration, intercepts all invocations to such methods (in the example, `method1`), and delegates their execution to the target middleware (in the example, Condor-G [49]) by means of an appropriate connector. From an architectural perspective, BYG provides a software tier that mediates between an ordinary Java application, or the client side, and Grid middlewares, or the server side. Gridified classes are run at the server side by means of connectors, whereas non-gridified classes remain at the client side. In principle, BYG can exploit any Grid middleware exposing a remote job submission interface for executing Java code.

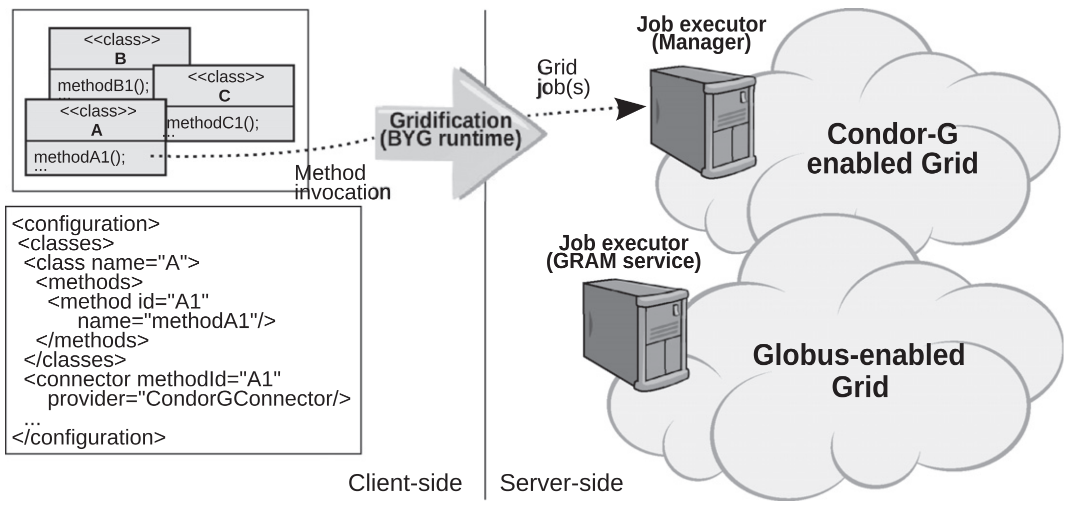
When configuring connectors, employing or not a specific Grid execution service such as Condor-G or Satin is mostly subject to availability factors, i.e. whether an execution service running on the target Grid is up and waiting for jobs. Furthermore, the choice of gridifying an individual operation depends on whether the oper-



<<class>> B
methodB1();
<<class>> C
methodC1();
<<class>> A
methodA1();
Java application (bytecode)
+
Gridification
Configuration (Grid middlewares to be used)
Unmodified class B
class C
Java Virtual Machine (JVM)
Grid-enabled class A
BYG runtime
Injected connector
Application layer (Gridified applications)
Condor-G
Satin
GridGain
Execution layer (Grid middlewares)
Infrastructure layer (the Computational Grid itself)

```
<configuration>
 <classes>
  <class name="A">
   <methods>
    <method id="A1"
         name="methodA1"/>
   </methods>
  </classes>
  <connector methodId="A1"
     provider="CondorGConnector/>
   ...
</configuration>
```

Fig. 1. An overview of BYG.



```
<configuration>
 <classes>
  <class name="A">
   <methods>
    <method id="A1"
         name="methodA1"/>
   </methods>
  </classes>
  <connector methodId="A1"
    provider="CondorGConnector/>
  ...
</configuration>
```

Fig. 2. Submitting ordinary Java methods as Grid jobs.

ation is suitable for execution on a Grid. The potential performance gains in gridifying an application are subject to two user design factors, namely the amount of data (i.e. parameter values) to be passed on to the gridified operations, and the computational requirements of such operations. In this sense, BYG alleviates the burden of adapting and submitting an ordinary application for execution on a Grid, while both factors (i.e. amount of data and computational requirements) must be estimated early by the user. Basically, this is similar to the analysis that must be carried out prior to introduce parallelism into any sequential code with technologies such as MPI or PVM in order to determine whether the code may actually benefit from being parallelized or not.

The implementation of BYG works by modifying bytecodes at runtime to delegate and submit the execution of certain application methods to external Grid execution services. BYG-enabling an application only requires the user to specify an XML file listing which methods are to be gridified and what Grid services (or platforms) are to be employed, and to add an argument to the Java Virtual Machine (JVM) program in the command that initiates the execution of the user application. BYG provides a connector for accessing the services of Satin [54], a Java-based framework for parallelizing applications on LANs and WANs. However, we are developing connectors for other Grid middlewares as well. This will allow users to take advantage of features not present in Satin such as graphical monitoring of running computations.

An initial stable release of BYG has been developed, which supports the functionality described in the rest of the paper on top of Satin version 2.1. The tool is open source and is available for download at http://www.exa.unicen.edu.ar/cmateos/projects.html. The next subsection focuses on providing a by-example explanation of the use of this tool. Please refer to [37] for a comprehensive discussion on the implementation of BYG.

3.1. Gridifying applications with the BYtecode Gridifier

To gridify a conventional application with BYG, it is necessary to supply a configuration file (XML [1] format), which lists both the application classes to be Grid-enabled and the Grid middlewares or execution services selected for execution. Particularly, users specify within this file the signature of the methods from these classes that are to be processed with BYG, and the binding information that depends on the node that plays the role of job executor of each service or middleware. A job executor is a frontend middleware-level component that resides on a specific Grid node, accepts jobs for execution and can be contacted by using various protocols. Examples of Grid job executors include the Manager component and the GRAM service of the Condor-G [49] and Globus [15] Grid platforms, respectively. Fig. 2 illustrates the notions exposed so far. As depicted, to gridify an application with BYG, the user must provide the following information:

1. The list of Java methods (owner class and signature) to be gridified. This information is enclosed within a <classes> element.
2. For each one of the above methods, the Grid execution service and consequently the connector to be used. This information is specified within a <connectors> element. Connectors are implemented through different classes that are shipped together with the BYG runtime.
3. For each one of the connectors, the IP address and the port of the Grid node that hosts the target job executor, and the desired job submission protocol from the set of the protocols supported by the job executor. For instance, the Manager component of Condor-G provides a socket-based job submission mechanism but also a submission interface based on Web Services [52]. The IP address, port and protocol binding information is placed within a <bindings> element.

For example, the following XML code gridifies the double integrate(double a, double b, double epsilon) method from the example.AdaptiveIntegration class by means of the job executor of the Condor-G middleware:

```
<configuration
xsi:noNamespaceSchemaLocation=‘‘byg.xsd’’
    xmlns:xsi=“http://www.w3.org/2001/XMLSchema-
instance”>
  <!-- Methods to gridify -->
  <classes>
    <class name=‘‘example.AdaptiveIntegration’’>
      <methods>
        <method id=‘‘mymethod’’ name=‘‘integrate’’>
          <parameter name=‘‘a’’ type=‘‘double’’/>
          <parameter name=‘‘b’’ type=‘‘double’’/>
          <parameter  name=‘‘epsilon’’  type=‘‘dou-
ble’’/>
        </method>
```

[1] A brief tutorial on XML can be found at http://www.w3schools.com/xml.

```
      </methods>
    </class>
</classes>
<!-- Connectors to use -->
<connectors>
  <connector                methodId=‘‘mymethod’’
bindingId=‘‘mybinding’’
      provider=‘‘org.isistan.byg.connectors.Con-
dorGConnector’’/>
      </connectors>
      <!-- Middleware-specific bindings -->
      <bindings>
      <binding id=‘‘mybinding’’ name=‘‘condor’’>
      <property        name=‘‘protocol’’>sockets</
property>
      <property       name=‘‘address’’>condor_man-
ager_ip_address</property>
        <property        name=‘‘port’’>condor_man-
ager_port</property>
      </middleware>
    </binding>
  </bindings>
</configuration>
```

Basically, BYG supports 1:N relationships between classes and methods (one or more methods of the same class can be gridified), N:1 relationships between methods and connectors (the same connector can be used for submitting different methods), and finally 1:1 relationships between connectors and bindings. In the above example, we have defined one connector responsible for submitting each invocation to the integrate method to the Condor-G Manager listening at [condor_manager_ip:condor_manager_port] by using socket-based communication. This bridging is performed by the CondorGConnector class from the BYG library, and the BYG core runtime, which injects this class into the compiled code of the AdaptiveIntegration application class so that, when executing the whole application, each call to integrate is submitted to Condor-G instead of executed locally.

To inject connector classes into ordinary ones, BYG relies on the support for agents provided by Java. A Java agent is a pluggable user-provided Java library that customizes the class loading process by performing bytecode transformations. This is, upon loading any application class, the JVM contacts (if defined) the corresponding Java agent and loads the bytecode resulted from passing the class through the agent. Fig. 3 shows the differences between running a Java application in the usual way, i.e. without Java agents (left), versus executing it by taking advantage of a Java agent (right). In the former case, both the user and the Java runtime class files are loaded and executed as is, whereas in the latter case a Java agent intercepts the class loading process and optionally modify user classes prior to execution.

Roughly, the BYG runtime is implemented as a Java agent. Precisely, the BYG agent dynamically modifies application classes to "talk" to the configured connectors to run the gridified methods of the application. Particularly, to BYG-enable our example application (i.e. to activate the BYG agent), the startup command that launches the user application must look like:

```
java -javaagent:byg.jar=<config-file>
   example.AdaptiveIntegration      [application
parameters]
```

The -javaagentswitch instructs the JVM to use the Java agent implemented by the byg.jar library. The characters enclosed within the "<" and ">" are the options for initializing the agent. Then, when the application starts, the BYG agent extracts from config-file the list of methods to gridify and their associated connectors, and then transforms the bytecodes of the methods as their owner classes are loaded by the JVM. To this end, BYG employs ASM [41], a small and fast Java-based bytecode manipulation framework.

Modifying an individual method involves two different tasks. First, its body is rewritten to include the instructions (or "stub") for delegating its execution to the connector class associated to the method (CondorGConnector in our case). The stub uses the corresponding binding information to submit the adapted version of the bytecode of the method for execution to the Grid every time this method is called by the application. Precisely, this adaptation represents the second task, given by the modification of the original bytecode of both the method and its owner class in order to be compliant to the bytecode anatomy prescribed by the target Grid middleware. Some platforms require applications to extend or to implement specific API classes, use certain API calls to carry out distribution and parallelism, and so on. Fig. 4 depicts an overview of the mechanism implemented by the BYG agent to dynamically

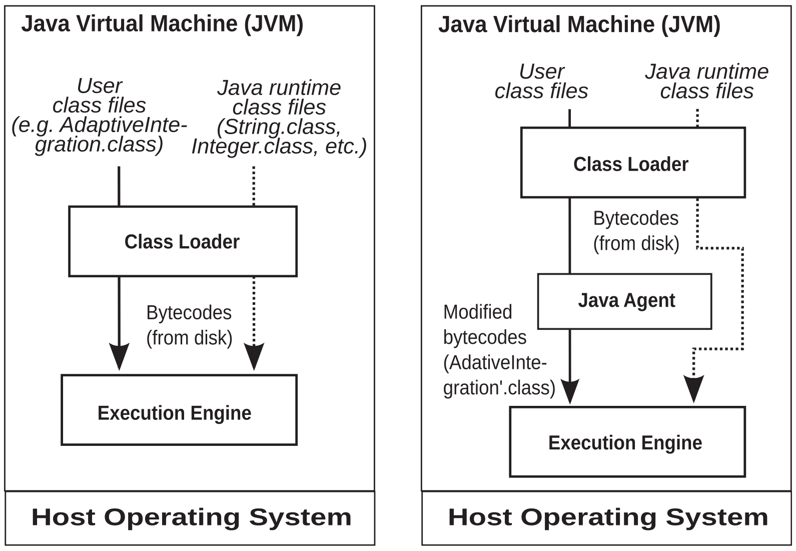
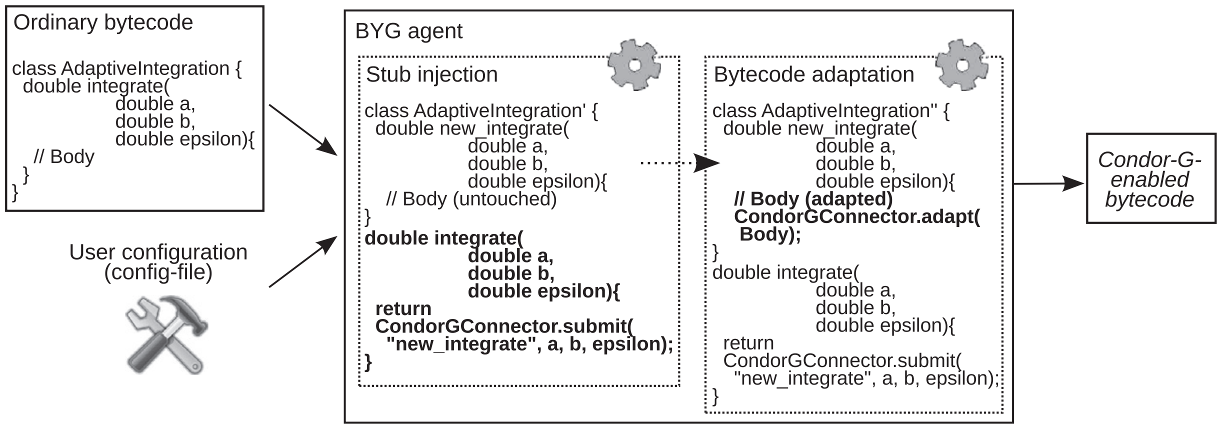

Java Virtual Machine (JVM)
User class files (e.g. AdaptiveIntegration.class)
Java runtime class files (String.class, Integer.class, etc.)
Class Loader
Bytecodes (from disk)
Execution Engine
Host Operating System
Java Virtual Machine (JVM)
User class files
Java runtime class files
Class Loader
Bytecodes (from disk)
Java Agent
Modified bytecodes (AdativeIntegration'.class)
Execution Engine
Host Operating System

Fig. 3. Modifying user classes on the fly: Java agents.



```
class AdaptiveIntegration {
  double integrate(
                double a,
                double b,
                double epsilon){
    // Body
  }
}
```

```
class AdaptiveIntegration' {
  double new_integrate(
                double a,
                double b,
                double epsilon){
    // Body (untouched)
}
double integrate(
                double a,
                double b,
                double epsilon){
  return
  CondorGConnector.submit(
    "new_integrate", a, b, epsilon);
}
```

```
class AdaptiveIntegration'' {
  double new_integrate(
                double a,
                double b,
                double epsilon){
    // Body (adapted)
    CondorGConnector.adapt(
     Body);
}
double integrate(
                double a,
                double b,
                double epsilon){
  return
  CondorGConnector.submit(
    "new_integrate", a, b, epsilon);
}
```

Fig. 4. Overview of the BYG agent.

obtain the Grid-enabled counterpart of an ordinary class such as AdaptiveIntegration.

The transformations performed at the second step (labeled in the figure as "Bytecode adaptation") strongly depends on the Grid middleware selected for connecting the input bytecode to a Grid [37]. For example, middlewares such as Condor-G, which rely on coarse-grained execution models that do not support parallelism within a method, do not require much transformations. Moreover, middlewares relying on a finer execution model and providing parallelism at the method level such as Satin makes the modification process more challenging. The next subsection focuses on explaining these notions in the context of the Satin platform, for which BYG provides a connector.

3.2. The Satin connector

Satin [54] is a framework for programming parallel divide and conquer Java applications on local-area and wide-area clusters. Satin provides programmatic mechanisms for indicating which methods of a sequential application are parallelized and synchronizing subcomputations. We have built a connector for this framework, which relieves developers from the burden of manually using the Satin API for parallelizing their applications by semi-automatically deriving a Satin-aware application from a sequential divide and conquer Java application. The next subsection explains the parallel programming model proposed by Satin. Section 3.2.2 presents an overview of our Satin connector.

3.2.1. Satin: Programming model

The divide and conquer model is an algorithm design technique that is based on implementing a problem by breaking them down into several subproblems of the same type, until trivial subproblems are obtained, which are in turn solved directly. The solutions to the different subproblems are then combined to build the solution to the whole problem. Most divide and conquer algorithms are then naturally implemented recursively, i.e. by issuing several recursive calls to the method implementing the problem. On the other hand, results of recursive calls are combined to give a solution to a larger problem.

Let us come back to the example AdaptiveIntegration class introduced so far. Now, let us suppose we provide a divide and conquer implementation for the integrate method, which computes the integral of a fixed function within a given interval (a, b). The integral value can be approximated by recursively dividing the input interval into two subintervals as long as the difference between the area of the trapezoid and the sum of the areas of the trapezoids of the subintervals is not smaller than some threshold epsilon, as follows:

```
 1 class AdaptiveIntegration {
 2   double function(double value){...}
 3   double integrate(double a, double b, double
   epsilon)
 4     double delta = ((b−a)/2);
 5     double total = delta
   * (function(a) + function(b));
 6     double left = (delta/2) * (function(a) + func-
   tion((b − a)/2 + a));
 7     double right = (delta/2) * (function(b) + func-
   tion((b − a)/2 + a));
 8     double diff = total − (left + right);
 9     if (diff < 0)
10       diff = −diff;
11     if (diff < epsilon)
12     return total;
13     double res1 = integrate((b − a)/
   2 + a, b, epsilon);
14 double res2 = integrate(a, (b − a)/2 + a, epsilon);
15     return res1 + res2;
16   }
17 }
```

Basically, the recursive calls to integrate of lines 13 and 14 are the *divide* phase of the algorithm, while lines 11-12 represent its *conquer* phase, i.e. the case when the problem at hand becomes small enough to be solved directly without further subdividing it.

The Satin programming model refines the sequential semantics of divide and conquer applications such as the one implemented by the above code to support parallelism in the divide phase. Specifically, Satin allows recursive calls to be solved in parallel to increase the performance of the algorithm by providing two primitives: an implicit one (spawn) to create parallel subcomputations, and an explicit one (sync), to programmatically block execution until subcomputations are finished. Methods considered for parallel execution must be included in the so-called *marker interfaces*, which are regular Java interfaces.

Let us parallelize our example application with Satin. To this end, we have to specify the method that is subject to parallel execution in a marker interface, which in turn must extend the satin.Spawnable interface from the Satin API:

```
interface AdaptiveIntegrationMarker extends
satin.Spawnable {
    double integrate(double a, double b, double epsilon);
```

and then modify our application to implement the newly generated marker interface and to extend the satin.SatinObject API class:

```
class AdaptiveIntegration extends satin.SatinObject
        implements AdaptiveIntegrationMarker{
    ...
}
```

Up to now, we have indicated Satin which methods of our application must be executed in parallel or, in other words, trigger independent parallel subtasks. However, we have to explicitly indicate in the application code the points in which it is necessary to wait for child computations to complete. This is like providing a join point or barrier that causes any task not to proceed and to wait for divide parts of the problem, whereupon the associated subresults are available and can be used to build a larger result. Returning to the example, the synchronized version of the integrate method is:

```
1 double integrate(double a, double b, double epsilon){
2   ...
3   double res1 = integrate((b − a)/2 + a, b, epsilon);
4   double res2 = integrate(a, (b − a)/2 + a, epsilon);
5   super. sync();
6   return res1 + res2;
7 }
```

As shown in the above code, at line 5, we have introduced a call to sync, which is the Satin synchronization primitive inherited from satin.SatinObject. This call prevents the application from combining subresults represented by yet-not-assigned variables. A practical rule for correctly using sync is to check that a call to this primitive is issued between the statements including recursive calls (i.e. lines 3 and 4) and those that access their results (i.e. line 6). It is worth noting that this analysis is trivial for the case of our example, but for applications involving more statements and complex control structures, it is tedious and significantly more error-prone.

In summary, after specifying the marker interface for the application, modifying the structure of the corresponding class and inserting appropriate synchronization calls into the application code, the developer must feed a special postprocessor provided by Satin with a compiled version of the application. This postprocessor translates the invocations to the divide and conquer method(s) listed in the marker interface (in our case integrate) into a Satin runtime task. In this way, at runtime, any call to this method will activate their associated task, whose execution is performed in parallel. Conceptually, this mechanism is similar to creating an independent thread for executing such recursive calls. Moreover, developers can configure Satin to exploit local and distributed clusters to execute such tasks or "threads", thus potentially improving the performance of the application.

3.2.2. Taking satin a step further

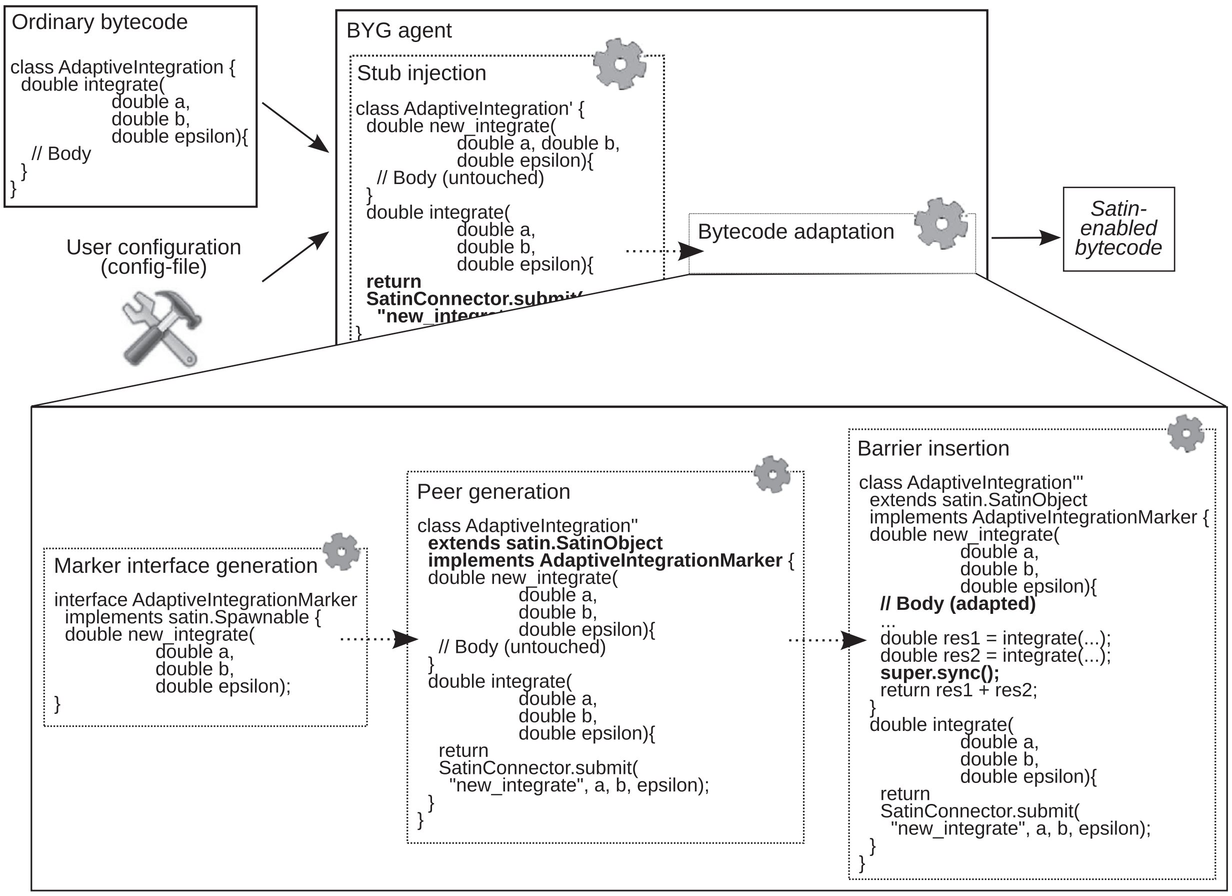
Our Satin connector semi-automatically reproduces the previous "satinification" tasks from a compiled, ordinary divide and conquer application that has not been explicitly coded to exploit the Satin API. Basically, the connector generates the marker interface based on the configuration of the application, and rewrites the bytecode of the corresponding class to extend/implement the necessary classes and interfaces and thus make it compliant to the Satin application structure. In addition, and more important, the connector inserts proper calls to sync by deriving a high-level representation from the bytecode and analyzing the points where barriers are needed. To execute the Satin-enabled version of applications, BYG relies on a software layer that wraps the Satin runtime. For more details on this extended Satin runtime, see [37].

Besides injecting instructions to execute ordinary methods on Satin (the "Stub injection" task in Fig. 4), the Satin connector dynamically adapts the bytecodes of both these methods and their owner classes to be compliant with the application anatomy prescribed by Satin. Basically, the connector carries out three main tasks:

- Marker interface generation: As explained, Satin requires applications to include a marker interface, which lists the methods considered for parallel execution. The Satin connector builds this interface from the methods listed in the XML configuration for the class being "satinified". The reader should recall that this information is included within the <classes>section of the configuration.
- Peer generation: Additionally, Satin applications must implement a marker interface and to extend from SatinObject. A clone (from now on *peer*) of the sequential class under consideration is created by the Satin connector and modified to fulfill these requirements.
- Barrier insertion: Based on an heuristic algorithm, the connector inserts calls to the Satin sync primitive at appropriate places of the spawnable methods of the peer. The heuristic aims at preserving the operational semantics of the (sequential) original algorithm while minimizing the calls to the primitive.

Fig. 5 depicts the steps performed by the connector to build the Satin-enabled version of an ordinary class. The connector builds the corresponding marker interface and a Satin peer from the class being processed. In a subsequent step, the Satin connector inserts Satin synchronization into the peer by using the heuristic algorithm. Afterwards, the peer is instrumented with the tools of the Satin platform. At runtime, the peer is instantiated and submitted for execution to the abovementioned extended Satin runtime by the ordinary application through the injected stub. To activate this behavior, the configuration file of the input application must be:

```
<configuration
xsi:noNamespaceSchemaLocation=''byg.xsd''
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance">
   <!-- Methods to gridify (same as before) -->
   ...
   <!-- Connectors to use -->
   <connectors>
   <connector              methodId=''mymethod''
bindingId=''mybinding''
      provider=''org.isistan.byg.connectors.Sat-
inConnector''/>
   </connectors>
   <!-- Middleware-specific bindings -->
   <bindings>
     <binding id=''mybinding'' name=''satin''>
     <property       name=''protocol''>sockets</
property>
     <property   name=''address''>satin_server_i-
p_address</property>
     <property  name=''port''>satin_server_port</
property>
   </binding>
  </bindings>
</configuration>
```



```
class AdaptiveIntegration {
  double integrate(
                  double a,
                  double b,
                  double epsilon){
    // Body
  }
}
```

```
class AdaptiveIntegration' {
  double new_integrate(
                  double a, double b,
                  double epsilon){
    // Body (untouched)
  }
  double integrate(
                  double a,
                  double b,
                  double epsilon){
  return
  SatinConnector.submit(
    "new_integr
}
```

```
interface AdaptiveIntegrationMarker
  implements satin.Spawnable {
  double new_integrate(
                  double a,
                  double b,
                  double epsilon);
}
```

```
class AdaptiveIntegration''
  extends satin.SatinObject
  implements AdaptiveIntegrationMarker {
  double new_integrate(
                  double a,
                  double b,
                  double epsilon){
    // Body (untouched)
  }
  double integrate(
                  double a,
                  double b,
                  double epsilon){
    return
    SatinConnector.submit(
      "new_integrate", a, b, epsilon);
  }
}
```

```
class AdaptiveIntegration'''
  extends satin.SatinObject
  implements AdaptiveIntegrationMarker {
  double new_integrate(
                  double a,
                  double b,
                  double epsilon){
    // Body (adapted)
    ...
    double res1 = integrate(...);
    double res2 = integrate(...);
    super.sync();
    return res1 + res2;
  }
  double integrate(
                  double a,
                  double b,
                  double epsilon){
    return
    SatinConnector.submit(
      "new_integrate", a, b, epsilon);
  }
}
```

Fig. 5. Satin-enabling ordinary bytecode: the Satin connector.

The algorithm for inserting barriers works by iterating the instructions of a method and detecting the points in which a local variable is either *defined* or *used* by a statement. A variable is defined when the result of a recursive call is assigned to it, whereas it is used when its value is read. To work properly Satin requires that statements can read such variables provided a sync has been previously issued. Then, our algorithm operates by modifying the bytecode to ensure a call to sync is done between the definition and use of a local variable, for any execution path between these two points. Moreover, as sync suspends the execution of the method until *all* subcomputations associated to defined variables have finished, our algorithm uses an heuristic to keep the correctness of the program while minimizing the inserted calls to sync for the sake of efficiency. It is out of the scope of this paper to discuss the internals of this heuristic algorithm. For details on this algorithm, please refer to [37].

4. Case studies

This section describes the experiments that were performed to empirically evaluate BYG. The contents of the section are a much more rigorous version of, and a complement to, the experiments reported in [37], in which we used classic CPU-intensive benchmark applications to compare BYG against Satin on a small LAN. To provide stronger evidence on the applicability of BYG, we measured the performance as well as resource usage that resulted from employing GridGain, Satin and BYG for parallelizing two real-world applications, specifically ray tracing (Section 4.1) and sequence alignment (Section 4.2), on a wide-area Grid. The goal of the evaluation was to determine whether the automatic approach to gridification followed by BYG is competitive compared to manual gridification when using GridGain or Satin with regard to the abovementioned aspects with realistic applications on a Grid setting. On one hand, we used the Satin platform so as to assess the differences between manually-generated Satin codes and Satin-enabled codes obtained by using BYG. On the other hand, we choose GridGain since it is a stable and healthy open source Grid platform that has recently became very popular for developing distributed applications.

First, we set up a LAN comprising 15 nodes running Mandriva Linux 2009.0, Java 5 and Satin 2.1 connected through a 100 Mbps network. We used 8 single core nodes with 2.80 MHz CPUs and 1.25 GB of RAM, and 7 single core nodes with 3 MHz CPUs and 1.5 GB of RAM. Then, we established a wide-area Grid on top of this LAN by employing WANem version 2.0 [48], a software for emulating WAN conditions over a local-area network. We emulated 3 remote clusters C_1, C_2 and C_3 by using 4, 5 and 6 of the nodes of the LAN, respectively, which were connected together by using virtual Internet links (see Fig. 6). Each WAN link was a T1 connection (i.e. a bandwidth of 1544 Mbps) with a round-trip latency of 160 ms and a jitter of 10 ms, therefore inter-cluster latencies were in the range of 150–170 ms. Particularly, these are network conditions commonly found in Internet-wide Grids.

For the sake of fairness, all tools were configured to use the load balancing algorithm that best fitted the experimental setting. On one hand, for the GridGain applications we employed its Round Robin load balancing with the default configuration, which according to the authors provides a fair distribution of tasks among the nodes of a Grid and therefore works well in most cases.[2] Basically, upon

[2] http://www.gridgainsystems.com/wiki/display/GG15UG/GridRoundRobinLoadBalancingSpi.

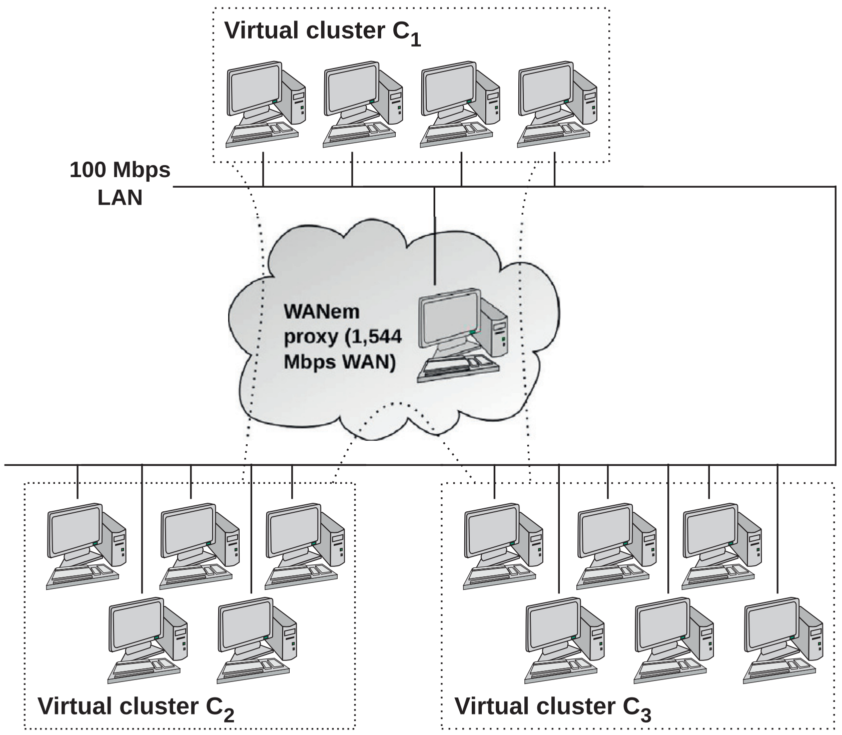


Fig. 6. Setting used in the experiments.

executing an application, the algorithm randomly picks a Grid node and then dynamically and sequentially assigns tasks for execution in a round-robin fashion. On the other hand, both the Satin and the BYG implementations of the ray tracing and sequence alignment applications were configured to take advantage of the Cluster-aware Random Stealing (CRS) [54] task scheduling algorithm provided by Satin. With CRS, when a Grid node becomes idle, it attempts to steal an unfinished task both from nodes belonging to the same local cluster or external nodes, however intra-cluster steals have a greater priority than inter-cluster ones, minimizing expensive WAN communication.

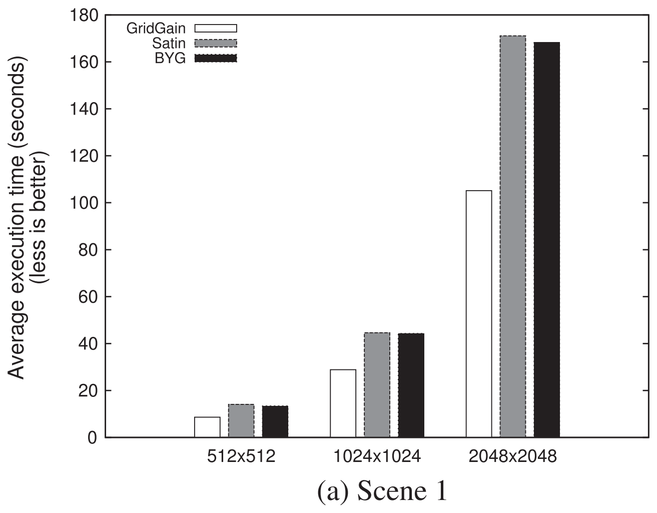
4.1. Ray tracing

Ray tracing is a widely-known rendering technique that generates a digital picture from an abstract description of a 3D scene [24]. Basically, we based our experiments on a parallel divide and conquer ray tracing algorithm from the Satin project,[3] which operates by deriving an initial image from the input scene, dividing this image to recursively apply the algorithm, and then joining the results to build the final picture. The BYG implementation was obtained by removing from the original Satin code any statement related to parallelism and/or tuning application execution to derive the sequential divide and conquer counterparts of the application. On the other hand, the GridGain implementation was obtained by altering the original Satin code to exploit the Google's map reduce parallel programming model [29], which is similar to the master-worker model and is supported by GridGain. We considered two variants of the application by altering the granularity of the runtime tasks, i.e. by splitting the image into 8×8 and 1×1 squares. In both cases, the algorithm first computes the correct color of each subimage and then reassembles the whole image. The second variant operates up to the pixel level, which allows the algorithm to output pictures with better quality but generates a larger number of tasks to execute at runtime.

To execute the three implementations of the first variant, we used two input scenes Scene 1 and Scene 2 (in NFF format [22]) with three different resolutions each (512×512, 1024×1024 and 2048×2048). Fig. 8 shows the resulting pictures for the largest resolution. On the other hand, Fig. 7 illustrates the average execution time of this variant for 60 runs. In all cases, standard deviations were in the range of 5–12%. Note that this percentage is somewhat high, however it is mainly explained by (a) the fact that GridGain used a random round robin load balancing support, (b) the fact that Satin and BYG relied on CRS for task scheduling, which implements a cluster-aware *random* task stealing algorithm, and (c) the variability inherent to WAN links in terms of bandwidth and latency. All in all, compared to Satin, BYG performed very well, considering that our goal is not to outperform existing Grid libraries but automating as much as possible their usage while achieving competitive performance. From Fig. 7a it can be seen that BYG performed similarly to Satin for the 1024×1024 image while incurred in an acceptable overhead of just few seconds for the other two. Fig. 7b shows that the performances for the more complex scene (Scene 2) were similar for the three resolutions. On the other hand, GridGain performed much better than Satin and BYG alike, which is explained by the less bureaucratic nature of its task distribution scheme. Unlike Satin and therefore BYG, in which each Grid node actively participates in the creation and assignment of parallel tasks, GridGain uses a master node that is in charge of distributing the tasks to the rest of the nodes. Then, the GridGain version of the ray tracing application performed better but as we explain next, it experienced an unfair assignment of parallel tasks to Grid nodes.

We measured the resource usage among the nodes of our simulated Grid by using the load average system metric of the Linux kernel, which is computed through an exponentially weighted moving average and is periodically stored in the "/proc/loadavg" file. Roughly, this metric allowed us to obtain the trend in CPU load

[3] http://www.cs.vu.nl/ibis/satin.html.



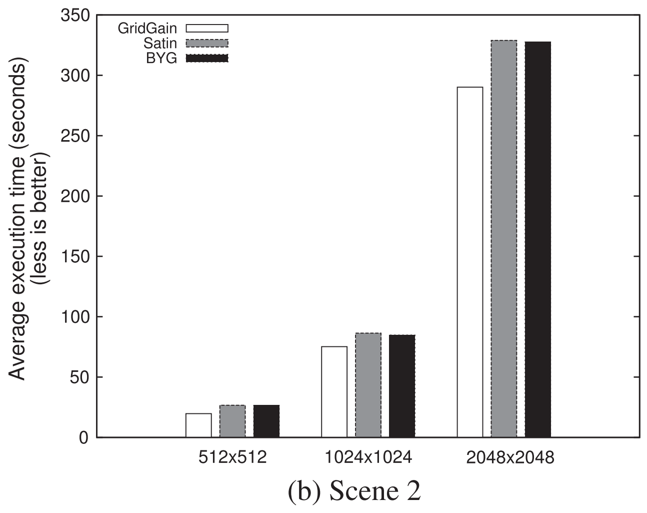


Fig. 7. Performance of the ray tracing application (task granularity = 8 × 8 squares).

(a) Scene 1

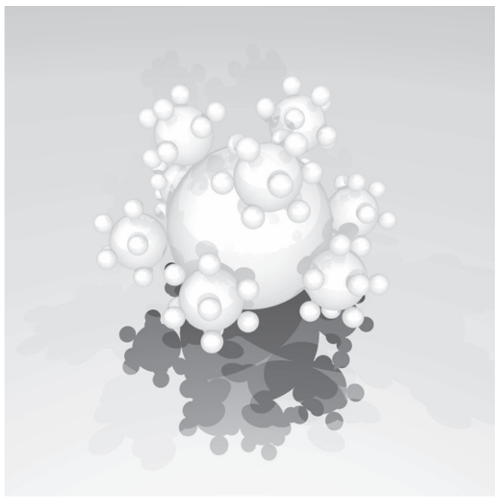

(b) Scene 2

Fig. 8. Grayscale version of the pictures resulted from executing the ray tracing application (resolution = 2048 × 2048 pixels).

at every single node of our Grid, so as to compute the standard deviation of these values to determine whether a node was more loaded than the others during the runs. In this sense, a small deviation is highly desirable, because it means that all nodes are evenly used, i.e. there are no nodes underused/overused. It is worth noting that CPU load is different from CPU utilization. Within a single node, this latter metric provides trend information of CPU usage but ignores the length of the queue maintaining the tasks waiting for taking possession of the CPU. For applications like ray tracing, which make extensive use of Grid resources, CPU utilization is always close to 100%. Hence, this metric is unable to accurately measure the load level of a node, which is in turn better reflected by the CPU load metric.

Table 1 shows the resulting fluctuation in CPU load, which was computed as the standard deviation of the load averages across nodes taken both at a 5-min and a 15-min window at the end of each test round. To obtain representative values, we introduced appropriate delays between each test battery in order to ensure that CPU load across nodes dropped down to zero. As the reader can see, the table shows that both Satin and BYG achieved similar and very uniform levels of CPU load among the Grid nodes. Conversely, GridGain experienced a rather uneven exploitation of the available resources. In summary, at least for this application and task granularity, GridGain achieved the best speedups but at the cost of performing a less fair usage of computational resources.

Fig. 9a and b shows the performance of the second variant of the application, i.e. the one using the smallest task granularity (sub-images of 1 × 1 pixels). Note that the Figure does not show the GridGain implementation, which proved to be too inefficient for this task granularity, registering overheads above 300% with respect to its competitors. Basically, using small granularities results in more runtime tasks to execute, which makes task scheduling more challenging to Grid middlewares. Although it cannot be generalized, this result in conjunction with the experiments illustrated in Fig. 7 may suggest that GridGain is better suited for applications with not-so-small task granularities. In contrast, Satin and BYG are designed to support efficient scheduling of parallel tasks irrespective of their granularity. Furthermore, in the present experiment BYG performed close to Satin, which is consistent with the results of Fig. 7. Both implementations achieved average load fluctuations in the range of 1–4% and 1–3% for the time windows of 5 and 10 min, respectively.

Table 1
Ray tracing: Fluctuation in CPU load (in percentage with regard to the average load across all nodes).

	Scene 1			Scene 2		
(5-Min)	512 × 512	1024 × 1024	2048 × 2048	512 × 512	1024 × 1024	2048 × 2048
GridGain	52	49	46	28	19	22
Satin	3	10	3	4	2	3
BYG	4	3	2	6	4	3
GridGain	26	31	39	28	18	16
Satin	2	2	2	2	6	1
BYG	3	1	1	4	3	2

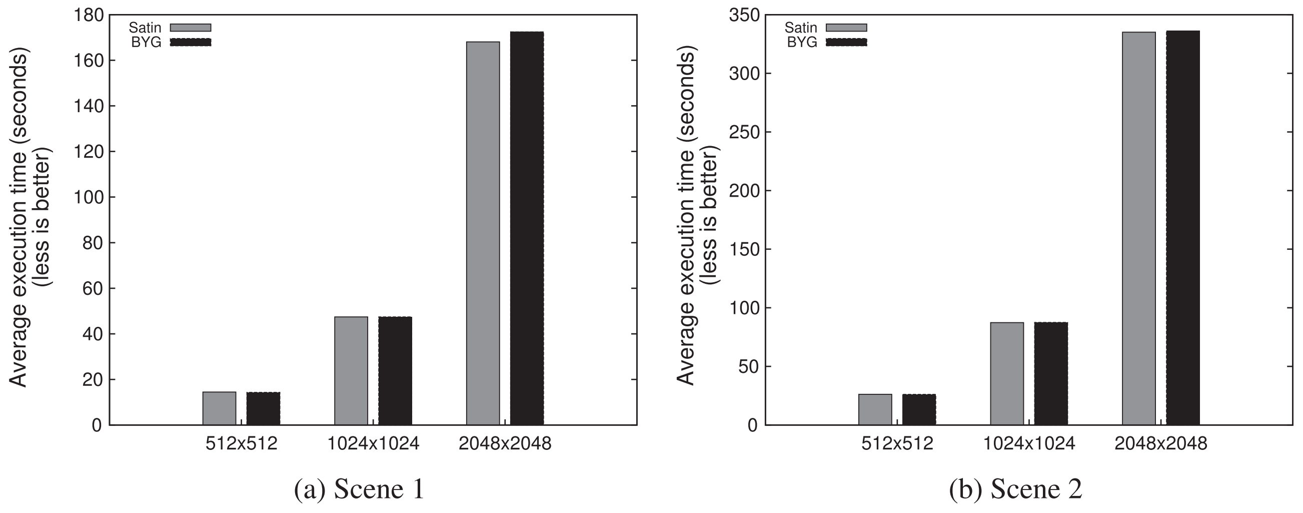


Fig. 9. Performance of the ray tracing application (task granularity = 1 × 1 squares).

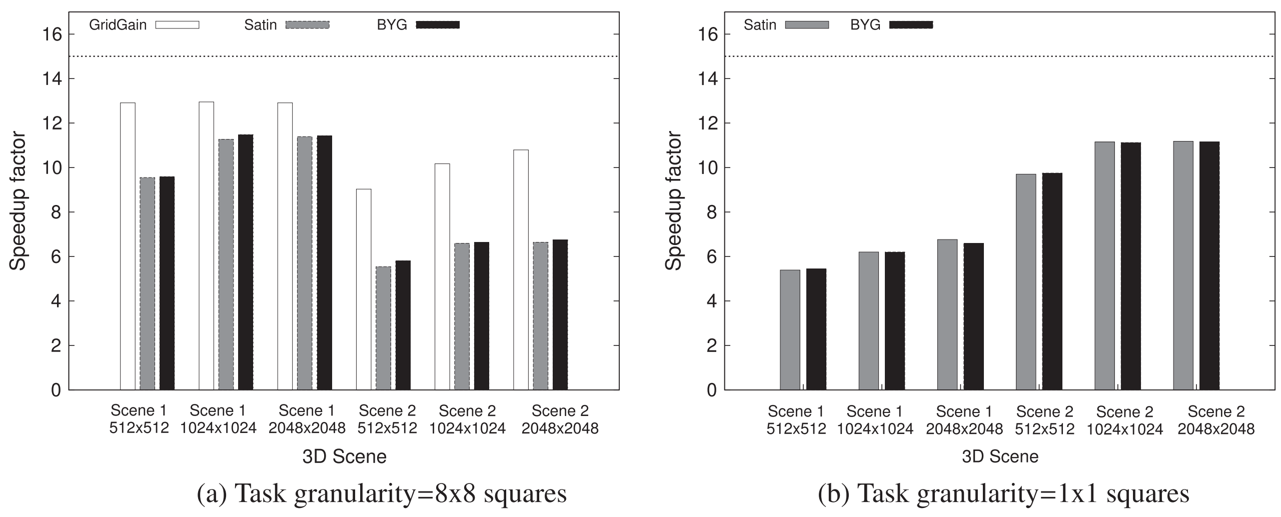


Fig. 10. Ray tracing: speedup factor. The upper dotted line represents the theoretical maximum.

As a complement, Fig. 10 shows the speedup factor achieved by the different implementations for the two task granularities. This factor was computed as T_{seq}/T_{grid}, where T_{seq} and T_{grid} are the times required to execute the sequential and gridified versions of these applications, respectively. To compute T_{seq}, the sequential ray tracing application was run on a node of the experimental setting with the best hardware capabilities in terms of CPU and memory.

4.2. Sequence alignment

The second test application was local pairwise sequence alignment,[4] a well-known problem in bioinformatics. The problem in-

[4] http://en.wikipedia.org/wiki/Sequence_alignment.

```
MASQGTKRSYEQMETDGERQNATEIRASVGRMIGGIGRFYIQMCTELKLNDYEGRLIQNSLTIERMVLSA
FDERRNKYLEEHPSAGKDPKKTGGPIYKRVDGKWVRELVLYDKEEIRRIWRQANNGDDATAGLTHIMIWH
SNLNDTTYQRTRALVRTGMDPRMCSLMQGSTLPRRSGAAGAAVKGVGTMVLELIRMIKRGINDRNFWRGE
NGRKTRIAYERMCNILKGKFQTAAQKAMMDQVRESRNPGNAEIEDLTFLARSALILRGSVAHKSCLPACV
YGPAVASGYDFEKEGYSLVGVDPFKLLQTSQVYSLIRPNENPAHKSQLVWMACNSAAFEDLRVSSFIRGT
RVLPRGKLSTRGVQIASNENMDAIVSSTLELRSRYWAIRTRSGGNTNQQRASAGQISTQPTFSVQRNLPF
DKATIMAAFSGNTEGRTSDMRAEIIKMMESARPEEVSFQGRGVFELSDERATNPIVPSFDMSNEGSYFFG
DNAEEYDN
```

Fig. 11. Sample protein sequence of the Influenza A H1N1 virus (obtained in England on December 31, 2007).

volves representing a biological entity such as a gene in a computer-understandable way (usually strings of characters) and manipulating the resulting representation by using sequence alignment algorithms. Fig. 11 shows, for instance, a sample protein sequence of the Influenza A H1N1 virus.

Basically, we took an existing parallel master–worker implementation of the application for aligning protein sequences. The source code of the application was obtained from the JPPF project[26]. Firstly, we derived the sequential version of this application and then we parallelized it back with GridGain, Satin and BYG. Furthermore, the original source code used JAligner [39], an open source library that implements an improved version of the Smith-Waterman algorithm [20]. Given any pair of sequences, the algorithm outputs a coefficient that represents the level of similarity between these two by using a scoring matrix from a set of predefined matrixes. To execute the experiments, we used the PAM120 matrix, which works very well in most cases.

The application aligned an unknown input sequence against an entire sequence database, which was replicated across the nodes of the experimental Grid to allow parallel tasks to locally access sequence data. The application operated by dividing the portions of the data to compare against into two different subproblems until a certain threshold on the data was reached. We used the same threshold values for GridGain, Satin and BYG. Moreover, we compared five different sequences against real-world protein sequence databases extracted from the National Center for Biotechnology Information (NCBI) Web site.[5] The NCBI is an organization devoted to computational biology that among other things maintains public genomes databases, disseminates biomedical information and develops bioinformatics software. It is worth mentioning that the sequences in the databases did not follow any special order (e.g. based on sequence size). Note that this allowed us to perform a fair evaluation in the sense that none of the evaluated middlewares and therefore their associated balancing strategies were favored over the others regarding input data ordering.

Concretely, we employed the sequence databases shown in Table 2. The last column of the Table indicates the number of generated parallel tasks as a consequence of using different thresholds. Basically, the larger the database, the finer the task granularity that was used, which enables for better parallelism. It is worth mentioning that the tests conceived the BYG implementation of the application as a mean to provide more evidence about the performance and resource usage of BYG compared to GridGain and Satin by using a realistic application. In this sense, the goal of these experiments it is not to come out with a better implementation of sequence alignment in Grid settings, for which specialized frameworks such as mpiBLAST [4] and G-BLAST [1] already exist.

Fig. 12 shows the average execution time for 60 runs of the different versions of the application, while Fig. 13 depicts the speedup factor. As the application is CPU-intensive but at the same time makes extensive use of sequence data, we did not achieve a high

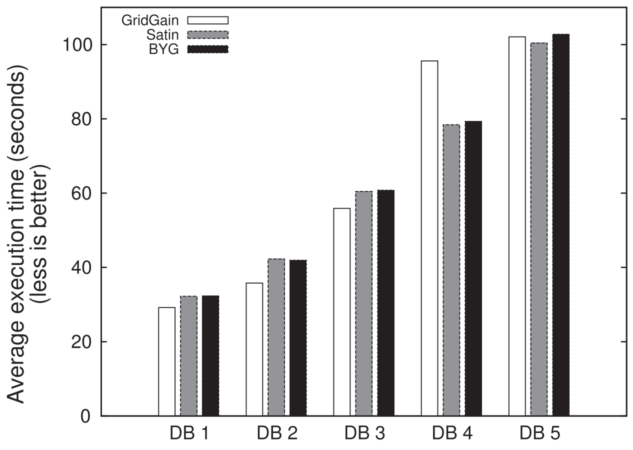


Fig. 12. Performance of the sequence alignment application.

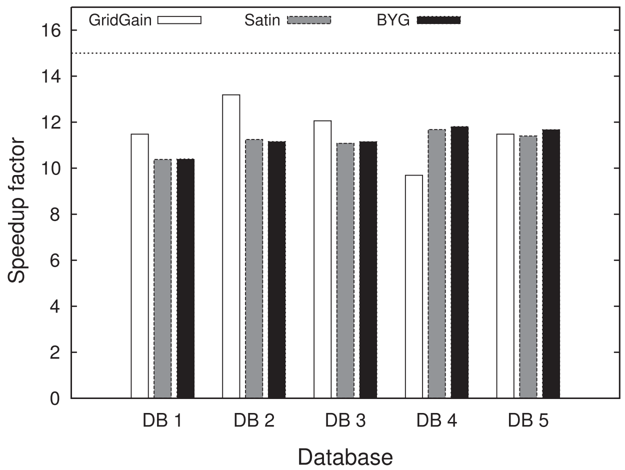


Fig. 13. Sequence alignment: speedup factor. The upper dotted line represents the theoretical maximum.

CPU utilization when aligning just one instance per run. In consequence, we decided to process two input target sequences simultaneously per execution. This resulted in CPU utilization close to 100% in the nodes of the experimental Grid, which in turn allowed us to measure resource usage through the CPU load metric in a representative manner. As illustrated in the figure, and similar to the case of the ray tracing application, BYG behaved close to Satin for all databases. On the other hand, standard deviations were in the range of 3–5% and 4–8% for the case of Satin and GridGain, respectively, and below 3% for the case of BYG. This fact may suggest that the execution time of the BYG variant was less affected by the data-intensive nature of the application, however this should be

[5] http://www.ncbi.nlm.nih.gov.

Table 2
Protein sequence databases used in the experiments.

Database	Size (# of sequences)	Size (MB)	Disease	Date	# of generated tasks
DB 1	4289	1.7	*Escherichia coli*	Unspecified	20
DB 2	4777	2.4	Influenza A (Human)	2009 outbreak (01/01/2009–31/06/2009)	20
DB 3	7672	3.5	Influenza B (Human)	All registered cases up to now	40
DB 4	9620	4.8	Influenza A (Human)	01/01/2007–12/31/2008	80
DB 5	12,325	6.2	Influenza A (Human)	01/01/2006–12/31/2008	80

Table 3
Sequence alignment: Fluctuation in CPU load (in percentage with regard to the average load across all nodes).

(5-Min)	DB 1	DB 2	DB 3	DB 4	DB 5
GridGain	44	34	25	27	28
Satin	3	3	3	3	3
BYG	3	3	4	4	3
(15-Min)	DB 1	DB 2	DB 3	DB 4	DB 5
GridGain	31	29	20	25	25
Satin	3	3	3	3	4
BYG	4	2	2	3	3

Table 4
Experimental results: qualitative analysis.

Middleware	Task granularity support	Delivered performance	Resource usage	Programmability
GridGain	Regular	Very good	Regular	Good
Satin	Very good	Good	Very good	Good
BYG	Very good	Good	Very good	Very good

further corroborated. Finally, Table 3 shows the fluctuation in average CPU load during the executions, which shows that BYG made a fair usage of Grid resources. Furthermore, the GridGain implementation steadily performed better than both Satin and BYG for the case of DB1, DB2 and DB3. On the other hand, the tests involving more runtime tasks resulted in execution overheads for the case of DB4 (see the dip in the bars) and marginal gains for DB5. Similarly to the ray tracing algorithm, GridGain had trouble in dealing with larger amounts of runtime tasks, while it slightly outperformed Satin and BYG with smaller ones.

4.3. Discussion

The experimental results reported in the previous subsections can be summarized according to several common dimensions of the problem of gridifying applications in the context of the studied platforms, namely task granularity support, delivered performance, resource usage and programmability [35]. The next paragraphs give some insights on these aspects in order to provide guidelines as to when to use each platform and when not. Table 4 qualitatively summarizes the obtained results in relation to these dimensions.

As mentioned, many Grid platforms work better when parallel tasks created as a consequence of parallelizing a sequential application have a coarse granularity. Under this scheme, the original problem is split into a small to medium number of tasks. On the contrary, other Grid platforms are oriented towards supporting finer tasks granularities. Therefore, their schedulers are designed to manage much larger number of tasks at runtime. In our experiments, the tests involving the usage of GridGain and fine granularities caused excessive overheads for the variant of the ray tracing application using 1x1 image squares, and resulted in performance losses for DB4 and DB5 in the sequence alignment application with respect to the rest of the databases.

Indeed, for sequence alignment, granularity could be made even finer by splitting the problem into more tasks, or alternatively processing DNA databases whose sequences are encoded by relying on a smaller alphabet compared to protein databases and thus local pairwise alignment at the sequence level becomes much less computationally intensive. By basing on the overheads resulted from using very fine task granularity for the case of the ray tracing application, we can reasonably extrapolate this result to say that GridGain would also perform in an inefficient way, rendering the comparison against Satin and BYG not viable. On the other hand, in the experiments, Satin and BYG proved to be more versatile with respect to task granularity. In fact, extensive recent experiments with benchmark applications and ultra fine task granularities [38] such as the ones that would result from feeding sequence alignment with DNA databases and/or increasing the number of parallel tasks that cooperatively align an input sequence confirm that BYG delivers good performance compared to Satin under such scenarios.

Moreover, for larger task granularities, GridGain outperformed its competitors, however it clearly made a less fair use of Grid resources. In addition, for more intensive experiments, the average fluctuation in resource usage tended towards smaller but still rather high values. For instance, the GridGain version of the ray tracing application experienced a fluctuation of 46% and 22% for Scene 1 and Scene 2, respectively. Likewise, the sequence alignment code had a fluctuation of 44% and 28% for DB1 and DB5, respectively. On the other hand, the fluctuation in resource usage for both test applications was in the range of 2–4% for the case of Satin, which delivered less performance but resulted in much better resource usage. All in all, the choice of whether to prioritize application performance over Grid resource usage in principle depends on the Grid setting being used. Specifically, bad resource usage may not be acceptable in Desktop Grid environments [8], which aim at arranging and taking advantage of idle CPU cycles of regular desktop PCs in use by individuals. However, in a dedicated Grid setting, a performance-oriented middleware like GridGain would suffice.

Furthermore, BYG behaved very competitively compared to Satin with respect to performance and resource usage, in spite of the fact that BYG adds some technological noise, i.e. our extended Satin runtime that handles the execution of transformed codes in parallel, and the heuristic for automating the process of inserting parallelism and synchronization into sequential applications. Intuitively, this should translate into rather different execution times. However, the experiments show that in some cases BYG certainly adds a performance overhead with respect to Satin, which we believe it is acceptable given the benefits of automatic parallelism in Grid environments to support scientists not proficient in distributed programming. In fact, the BYG versions of the two test applications did not require to make explicit usage of API-specific code, which in turn positively impacted on the size of the implementation code of the parallel applications. In this sense, Table 5 shows the TLOC values for the test applications, i.e. the total lines of code without considering neither blank nor comment lines. The Table also shows the GLOC metric [36], which counts developer-supplied lines explicitly invoking Grid API primitives within the application

Table 5
Test applications: code metrics.

Application	Middleware	TLOC	GLOC
Ray tracing	GridGain	1176	33
	Satin	1065	4
	BYG	1057	0
Sequence alignment	GridGain	581	44
	Satin	503	5
	BYG	493	0

code. As illustrated, BYG is based on an automatic approach to parallelism and thus isolates users from API-related details. In contrast, more verbose, API-oriented gridification platforms such as GridGain require users to know details of its programming library.

The differences obtained in T_{grid} for BYG and Satin are mostly explained by the places of the application code in which the calls to sync are located. Naturally, these differences stem from the fact that the Satin versions of the applications were parallelized and provided with synchronization by hand, while the BYG counterparts were parallelized by applying our heuristic on the sequential codes, which precisely attempt to reproduce the parallelization and synchronization tasks a human programmer would manually perform. These results show that BYG certainly simplifies the usage of parallel libraries like Satin without incurring in an excessive performance penalty and thus achieving competitive speedups. In other words, BYG stays competitive compared to directly using Satin, which is explained by the effectiveness of our generic synchronization heuristic. This claim is not done in isolation, but elaborated on the grounds of the experiments reported in this paper as well as similar results achieved with BYG in a cluster environment [37].

5. Conclusions and future work

This paper described BYG (BYtecode Gridifier), a new software tool to easily port ordinary compiled Java applications to Computational Grids. BYG lets users to Grid-enable existing Java applications by indicating which parts of their bytecode should run on a Grid without requiring programming effort and otherwise using configuration external to the application. To this end, BYG is based on novel bytecode rewriting techniques through which ordinary bytecodes are semi-automatically furnished with parallelism to exploit Grids. BYG targets Java applications implemented under the popular and versatile divide and conquer programming model. We can thus reasonably expect the tool will benefit a large number of today's applications.

At present, BYG is implemented on top of Satin, a framework that supports execution of applications on LANs and WANs. We evaluated BYG by gridifying two popular real-world resource-intensive applications, namely ray tracing and sequence alignment, by using both Satin and BYG on a wide-area Grid. Results show that most of the BYG versions performed very similarly to their Satin counterparts, and thus achieved very competitive speedups and resource usage. We believe this is an interesting result considering that BYG automates the use of Satin without incurring in performance overheads or unfair usage of Grid nodes to gridify applications. In addition, we also compared GridGain and BYG, in order to provide a wider spectrum of Java-based gridification tools and particularly to discuss the applications and Grid settings for which our tool is beneficial. In this sense, we concluded Section 4 by providing practical guidelines regarding when to use each tool and when not.

It is worth emphasizing that our approach does not aim at replacing explicit distributed and parallel programming models, such as the ones promoted by GridGain and Satin. Its utmost goal is to target users who would need to rapidly turn their sequential applications into parallel ones while dealing with as little parallel exploitation problems as possible. Basically, BYG addresses this requirement by advocating an automatic approach to support the process of obtaining a Grid-aware application. However, it is a well-known fact in parallel programming that such an implicit approach to parallelism may, in general, produce parallel applications whose performance is below the levels reached by using explicit parallelism [16], in which the developer has more control over the parallel behavior of their applications. In the context of our work, this means that using BYG may not necessarily lead to exploiting parallelism in an optimal way. As shown, BYG effectively inserts parallelism into sequential codes in a best-effort and heuristic way, leading to competitive speedups for fine-grained applications while offering adequate support for users with limited knowledge on parallel and Grid concepts.

At present, we are extending BYG into several directions. Particularly, we are refining its bytecode rewriting techniques to recognize some other high-level Java statements (e.g. try/catch and switch/case) and to optimize the insertion of Satin barriers. It is worth noting that this is not a limitation of our parallelization and synchronization algorithm but of its current implementation. We are also investigating how to generalize and support our gridification approach for other target Grid middlewares apart from Satin, and even other languages for scientific computing. On one hand, a prototype implementation of BYG on top of the Condor-G [49] middleware is underway. On the other hand, for gridifying binary codes obtained through *compiled* languages, a technological alternative is to employ Dyninst [50], an API that allows on-the-fly modification of native binary codes. This will allow us to adapt the ideas behind the BYG agent for introducing parallelism into the binary code produced by widely-adopted languages such as C and C++. Also, and similarly to the work by Papadimitriou and Terzidis [43], we are investigating how to integrate BYG with the Java scripting API [42], which allows developers to execute scripts implemented in various *interpreted* languages (e.g. Python, Ruby, BeanShell, etc.) within the Java runtime. Interestingly, this would greatly simplify the adoption of BYG as most of these interpreted languages are commonplace in scientific programming.

Finally, we will conduct experiments with BYG in larger Grid settings. We are working on the gridification of the ray tracing and the sequence alignment application on a real (i.e. not emulated) high-speed wide-area Grid. The infrastructure is a result of a country-wide Grid initiative of the Argentinian government that will connect several academic clusters across different provinces of Argentina, which was launched recently.[6] In addition, we will study the applicability of BYG to other domains, particularly finite element analysis. As a starting point, we will gridify the SOGDE 2D and 3D solver [17], which has been used for simulating tension tests in metals [18].

Acknowledgments

We thank the anonymous referees for their comments to improve the paper. We also thank Cristian Clasadonte for his valuable help managing the computing infrastructure required for conducting the experiments described in this paper. Finally, we acknowledge the financial support provided by ANPCyT through Grants PAE-PICT 2007-02311 and PAE-PICT 2007-02312.

References

[1] Afgan E, Bangalore P. Dynamic BLAST – a Grid enabled BLAST. Int J Comput Sci Network Security 2009;9(4):149–57.
[2] Alonso JM, Hernández V, Moltó G. GMarte: Grid middleware to abstract remote task execution. Concurr Comput: Practice Exper 2006;18(15):2021–36.
[3] Alonso JM, Hernández V, Moltó G. A high-throughput application for the

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

dynamic analysis of structures on a grid environment. Adv Eng Softw 2008;39(10):839–48.
[4] Archuleta J, Feng W-C, Tilevich E. A pluggable framework for parallel pairwise sequence search. In: 29th Annual international conference of the IEEE – Engineering in Medicine and Biology Society (EMBS '07); 2007. p. 127–30.
[5] Baduel L, Baude F, Caromel D, Contes A, Huet F, Morel M, et al. Grid computing: software environments and tools. In: Programming, composing, deploying on the grid. Berlin, Heidelberg, and New York: Springer; 2006. p. 205–29.
[6] Baitsch M, Li N, Hartmann D. A toolkit for efficient numerical applications in Java. Adv Eng Softw 2010;41(1):75–83.
[7] B.S. Center. Alya system – large scale computational mechanics; 2009. <http://www.bsc.es/plantillaA.php?cat_id=552> [accessed January 2011].
[8] Choi S, Kim H, Byun E, Baik M, Kim S, Park C, et al. Characterizing and classifying desktop grid. In: 7th IEEE international symposium on cluster computing and the grid (CCGRID '07), Rio de Janeiro, Brazil. Washington (DC, USA): IEEE Computer Society; 2007. p. 743–8.
[9] da Silva Cunha CA, Ferreira Sobral JL. An annotation-based framework for parallel computing. In: 15th Euromicro conference on parallel, distributed, and network-based processing (PDP '07), Naples, Italy. Los Alamitos (CA, USA): IEEE Computer Society; 2007. p. 113–20.
[10] Danelutto M, Pasin M, Vanneschi M, Dazzi P, Laforenza D, Presti L. PAL: exploiting Java annotations for parallelism. In: Achievements in European research on grid systems. United States: Springer; 2008. p. 83–96.
[11] Doolin DM, Dongarra J, Seymour K. JLAPACK – compiling LAPACK Fortran to Java. Scient Program 1999;7(2):111–38.
[12] Drummond LA, Galiano V, Migallón V, Penadés J. Interfaces for parallel numerical linear algebra libraries in high level languages. Adv Eng Softw 2009;40(8):652–8.
[13] Eyheramendy D. Innovation in engineering computational technology. In: High abstraction level frameworks for the next decade in computational mechanics. Saxe-Coburg Publications; 2006. p. 41–61.
[14] Foster I. The grid: computing without bounds. Scient Am 2003;288(4):78–85.
[15] Foster I. Globus toolkit version 4: software for service-oriented systems. J Comput Sci Technol 2006;21(4):513–20.
[16] Freeh VW. A comparison of implicit and explicit parallel programming. J Parallel Distrib Comput 1996;34(1):50–65.
[17] García Garino C. Un modelo numérico para el análisis de sólidos elastoplásticos sometidos a grandes deformaciones. PhD thesis. Barcelona (Spain): E.T.S. Ingenieros de Caminos, Universidad Politécnica de Catalunya; 1993.
[18] García-Garino C, Gabaldón F, Goicolea JM. Finite element simulation of the simple tension test in metals. Fin Elem Anal Des 2006;42(13):1187–97.
[19] Gonçalves RC, Ferreira Sobral JL. Pluggable parallelisation. In: 18th ACM international symposium on high performance distributed computing (HPDC '09), Garching, Germany. New York (NY, USA): ACM Press; 2009. p. 11–20.
[20] Gotoh O. An improved algorithm for matching biological sequences. J Mol Biol 1982;162(3):705–8.
[21] GridGain Systems. The GridGain open cloud platform; 2009. <http://www.gridgain.com> [accessed January 2011].
[22] Haines E. Neutral file format (NFF); 1992. <http://local.wasp.uwa.edu.au/pbourke/dataformats/nff/nff1.html> [accessed January 2011].
[23] Harbulot B, Gurd JR. Using AspectJ to separate concerns in parallel scientific Java code. In: 3rd International conference on aspect-oriented software development (AOSD '04), Lancaster, UK. New York (NY, USA): ACM Press; 2004. p. 122–31.
[24] Heckbert P, Haines E. A ray tracing bibliography. In: Glassner A, editor. Introduction to ray tracing. Academic Press, Inc.; 1989. p. 295–303.
[25] Hernández E, Cardinale Y, Pereira W. Extended mpiJava for distributed checkpointing and recovery. In: Recent advances in parallel virtual machine and message passing interface. Lecture notes in computer science, vol. 4192. Berlin/Heidelberg: Springer; 2006. p. 158–65.
[26] JPPF. JPPF Home; 2009. <http://www.jppf.org> [accessed January 2011].
[27] Jugravu A, Fahringer T. JavaSymphony, a programming model for the Grid. Fut Generation Comput Syst 2005;21(1):239–46.
[28] Kiczales G, Lamping J, Menhdhekar A, Maeda C, Lopes C, Loingtier J-M, et al. Aspect-oriented programming. In: Akşit M, Matsuoka S, editors. 11th European conference on object-oriented programming (ECOOP '97). Lecture notes in computer science, vol. 1241. New York (NY, USA): Springer; 1997. p. 220–42.
[29] Lämmel R. Google's MapReduce programming model – revisited. Sci Comput Program 2007;68(3):208–37.
[30] Laskowski E, Tudruja M, Olejnik R, Toursel B. Byte-code scheduling of Java programs with branches for desktop grid. Fut Generation Comput Syst 2007;23(8):977–82.
[31] Lee EA. The problem with threads. Computer 2006;39(5):33–42.
[32] Mackie RI. Design and deployment of distributed numerical applications using .NET and component oriented programming. Adv Eng Softw 2009;40(8):665–74.
[33] Maia PM, Mendonca NC, Furtado V, Cirne W, Saikoski K. A process for separation of crosscutting Grid concerns. In: ACM symposium on applied computing (SAC '06), Dijon, France. New York (NY, USA): ACM Press; 2006. p. 1569–74.
[34] Manolescu D, Beckman B, Livshits B. Volta: developing distributed applications by recompiling. IEEE Softw 2008;25(5):53–9.
[35] Mateos C, Zunino A, Campo M. A survey on approaches to gridification. Softw: Practice Exper 2008;38(5):523–56.
[36] Mateos C, Zunino A, Campo M. On the evaluation of gridification effort and runtime aspects of JGRIM applications. Fut Generation Comput Syst 2010;26(6):797–819.
[37] Mateos C, Zunino A, Campo M, Trachsel R, Programming Parallel. Models and applications in grid and P2P systems. In: An approach to just-in-time gridification of conventional Java applications. Advances in parallel computing. Amsterdam (The Netherlands): IOS Press; 2009. p. 232–60 [chapter BYG].
[38] Mateos C, Zunino A, Trachsel R, Campo M. A novel mechanism for gridification of compiled Java applications. Comput Inform, in press.
[39] Moustafa A. JAligner: open source Java implementation of Smith–Waterman; 2008. <http://jaligner.sourceforge.net> [accessed January 2011].
[40] National Science Foundation. ScaLAPACK; 2007. <http://www.netlib.org/scalapack> [accessed January 2011].
[41] ObjectWeb Consortium. ASM; 2009. <http://asm.objectweb.org> [accessed January 2011].
[42] Oracle Inc. javax.script (Java Platform SE 6); 2009. <http://download.oracle.com/javase/6/docs/api/javax/script/package-summary.html> [accessed January 2011].
[43] Papadimitriou S, Terzidis K. jLab: integrating a scripting interpreter with Java technology for flexible and efficient scientific computation. Comput Lang Syst Struct 2009;35(3):217–40.
[44] Ropo M, Westerholm J, Dongarra J. Recent advances in parallel virtual machine and message passing interface. In: Proceedings of the 16th European PVM/MPI users' group meeting, Espoo, Finland, September 7–10, 2009. Lecture notes in computer science. Berlin/Heidelberg: Springer-Verlag; 2009.
[45] Seymour K, Dongarra J. Automatic translation of Fortran to JVM bytecode. Concurr Comput: Practice Exper 2003;15(3–5):207–22.
[46] Shafi A, Carpenter B, Baker M. Nested parallelism for multi-core HPC systems using Java. J Parallel Distrib Comput 2009;69(6):532–45.
[47] Shafi A, Carpenter B, Baker M, Hussain A. A comparative study of Java and C performance in two large-scale parallel applications. Concurr Comput: Practice Exper 2009;21(15):1882–906.
[48] TATA Consultancy Services. WANem; 2008. <http://wanem.sourceforge.net> [accessed January 2011].
[49] Thain D, Tannenbaum T, Livny M. Condor and the grid. In: Berman F, Fox G, Hey A, editors. Grid computing: making the global infrastructure a reality. New York (NY, USA): John Wiley & Sons; 2003. p. 299–335.
[50] University of Maryland. Dyninst api; 2009. <http://www.dyninst.org> [accessed January 2011].
[51] University of Virginia. jPVM; 1999. <http://www.cs.virginia.edu/ ajf2j/jpvm.html> [accessed January 2011].
[52] Vaughan-Nichols SJ. Web services: beyond the hype. Computer 2002;35(2):18–21.
[53] Wang L, Jie W. Towards supporting multiple virtual private computing environments on computational grids. Adv Eng Softw 2009;40(4):239–45.
[54] Wrzesinska G, van Nieuwport R, Maassen J, Kielmann T, Bal H. Fault-tolerant scheduling of fine-grained tasks in grid environments. Int J High Perf Comput Appl 2006;20(1):103–14.
[55] Zhang B-Y, Yang G-W, Zheng W-M. JCluster: an efficient Java parallel environment on a large-scale heterogeneous cluster. Concurr Comput: Practice Exper 2006;18(12):1541–57.
[56] Zhang H, Lee J, Guha R. VCluster: a thread-based Java middleware for SMP and heterogeneous clusters with thread migration support. Softw: Practice Exper 2008;38(10):1049–71.