Asociación Argentina
de Mecánica Computacional

Mecánica Computacional Vol XXVIII, págs. 253-272 (artículo completo)
Cristian García Bauza, Pablo Lotito, Lisandro Parente, Marcelo Vénere (Eds.)
Tandil, Argentina, 3-6 Noviembre 2009

A TOOL FOR AUTOMATIC PARALLELIZATION OF CPU-INTENSIVE JAVA APPLICATIONS ON DISTRIBUTED ENVIRONMENTS

Cristian Mateos[a,b], **Alejandro Zunino**[a,b] **and Marcelo Campo**[a,b]

[a]*ISISTAN - UNCPBA. Campus Universitario, Tandil (B7001BBO), Buenos Aires, Argentina. Tel.: +54 (2293) 439682. Fax: +54 (2293) 439681.*

[b]*CONICET (Consejo Nacional de Investigaciones Científicas y Técnicas)*

Keywords: Grids, gridification, automatic parallelism, CPU-intensive applications, Java.

Abstract. Grid Computing, a relatively new paradigm for distributed computing, delivers the necessary computational infrastructure –the so-called Computational Grids– 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 conventional software implementing such problems to take advantage of Grid resources, which requires knowledge on Grid programming. The recent notion of gridifying conventional applications, which is based on semi-automatically deriving the Grid-aware version from the compiled code of an ordinary application, promises users to be relieved from the requirement of manual usage of Grid APIs within their codes. This paper describes a novel Java-based gridification tool, which allows users to automatically parallelize applications on local-area and wide-area Grids. Experiments confirm that BYG effectively exploits such Grids while delivers competitive performance with respect to manually using Grid APIs to gridify conventional software.

1 INTRODUCTION

Computational Grids are distributed heterogeneous clusters that allow scientists to build applications that demand by nature a huge amount of resources such as CPU cycles and memory Foster (2003). Examples of such applications include financial modeling, weather prediction, catastrophe simulation, aerodynamic design, drug discovery, amongst many others. Unfortunately, taking advantage of Computational Grids involves significant development effort and requires knowledge on distributed and 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, the 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).

The traditional approach to cope with the problem of easily exploiting Grids is based on employing programming APIs such as MPI Ropo et al. (2009) and PVM Ropo et al. (2009), which provide a standard and simple interface to Grids through the provision of primitives to execute parts of an application in a distributed and coordinated way. To this end, a developer must in principle indicate which parts of its application can benefit from being parallelized by including in the sequential code of his application appropriate calls to such primitives. Interestingly, APIs like MPI and PVM mitigate the complexity inherent to writing Grid applications. However, such APIs still require users to be proficient in parallel and distributed programming.

Recently, the notion of "gridifying" sequential applications Mateos et al. (2008) has appeared as an exciting approach for rapidly, seamlessly developing and executing applications in Computational Grids. Gridification tools seek to avoid the manual usage of parallel and distributed APIs within the code of user applications by automatically deriving the Grid counterparts from the (sequential) compiled code of these applications. In this article, we describe a new gridification tool for Java called BYG (BYtecode Gridifier), which operates by using some novel techniques for modifying bytecodes, this is, the binary code generated by the Java compiler. Basically, the current version of BYG targets Java applications implemented 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.

We evaluated BYG by gridifying and running several CPU-intensive classic divide and conquer applications on both a local-area and a wide-area Grid. In the experiments, we used the Satin Wrzesinska et al. (2006) Grid scheduler, which is designed for efficiently executing recursive applications on clusters and Grids. The results suggests that our approach offers an efficient and effective alternative to the problem of easy parallelization of sequential applications. Given the ever increasing popularity of the Java

language for distributed programming, which is mostly explained by its platform-neutral bytecode and its very competitive performance in large-scale distributed environments compared to traditional languages Shafi et al. (2009b), and the simplicity and versatility of the divide and conquer model, BYG is useful for gridifying a broad range of compute intensive sequential applications.

The next section discusses the most relevant related works. Section 3 overviews BYG and explains how it improves over them. For the most part, the Section describes the use of BYG in the context of the Satin Wrzesinska et al. (2006) Grid middleware. Section 4 reports the experiments that were carried out to evaluate BYG. Section 5 concludes the paper and discusses future works.

2 LITERATURE REVIEW

The problem of simplifying the development of high-performance scientific applications has been commonly addressed by either providing domain-specific solutions or general-purpose tools. The first approach offers APIs and runtime supports for using widely-used scientific libraries from within applications, whereas the second one allows users to write applications while not necessarily relying on specific scientific libraries.

Among the efforts in the former group is Baitsch et al. (2009), a Java toolkit for numerical intensive applications that provides Java wrappers to access numerical Fortran libraries such as BLAS and LAPACK. The toolkit also provides a Java-based library with classes for common vector, matrix and linear algebra operations. Similarly, f2j Seymour and Dongarra (2003) is a Fortran-to-Java translator specially designed to obtain the Java counterpart of the Fortran code of BLAS and LAPACK. Moreover, the jLab environment Papadimitriou and Terzidis (2009) offers a scripting language similar to Matlab and Scilab on top of Java. jLab supports the basic programming constructs of Matlab (e.g. operators for manipulating matrixes). Moreover, Eyheramendy (2006) proposes a Java-based framework for Computational Fluid Dynamics applications, which currently supports different finite elements formulations for basics mechanical problems, where some of them can be parallelized by using multiple threads.

Indeed, the idea of providing domain-specific tools is not only circumscribed to Java, as evidenced by similar supports for other programming languages. Recent examples are PyScaLAPACK Drummond et al. (2009), a Python interface to ScaLAPACK National Science Foundation (2007), this latter being a parallel implementation of the LAPACK linear algebra routines. Moreover, Mackie (2009) proposes a FEM distributed solver written in .NET. However, the two negative characteristics of the efforts following this approach is that they strongly restrict the kind of applications that can be written and, except for few cases, they are not capable of exploiting clusters and Grids.

Precisely, MPI and PVM are the oldest standards for building general-purpose parallel applications, which are parallelized by decomposing them into distributed components that communicate via message exchange. Many Java bindings for MPI (e.g. mpi-

Java Hernández et al. (2006), MPJ Express Shafi et al. (2009a)), PVM (e.g. jPVM University of Virginia (1999)) or both (JCluster Zhang et al. (2006)) exists. However, MPI and PVM are too low-level and thus require solid knowledge on parallel programming concepts from users Lee (2006). 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.

ProActive Baduel et al. (2006) is a Java API that provides *technical services*, a flexible support to address non-functional Grid concerns (e.g. object distribution and load balancing) by plugging external configuration to applications at deployment time. ProActive features integration with various Grid schedulers and supports execution of Scilab scripts on clusters and Grids. JavaSymphony Jugravu and Fahringer (2005) is a platform offering a semi-automatic execution model that deals with parallelism and load balancing of Grid applications, but also allows programmers to control such features via API calls. Unfortunately, using these API-inspired parallelization tools unavoidably requires to learn and use their associated APIs to parallelize applications, which is difficult to achieve for an average programmer.

Some tools aimed at further simplifying the complexity of the exposed parallel API and thus reducing this learning curve have been proposed. VCluster Zhang et al. (2008) supports execution of thread-based Java applications on multicore clusters by relying on a thread migration technique that achieves efficient dynamic load balancing. Similarly, DG-ADAJ Laskowski et al. (2007) offers execution of multi-threaded Java applications on desktop PC Grids. DG-ADAJ automatically derives graphs from the compiled bytecode of a Java application that 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 has been much criticized Lee (2006) due to the non-deterministic nature of thread execution, which makes application development rather difficult Lee (2006).

In this sense, Satin Wrzesinska et al. (2006) avoids the usage of threads by targeting recursive applications and modifying the compiled code of an application to handle the execution of recursive calls in parallel on a Grid. The user must indicate in the code the points in which forks (i.e. recursive calls) or joins (i.e. waiting for child computations) should take place. A similar framework for .NET is Volta Manolescu et al. (2008), which recompiles executable .NET applications on the basis of developer code annotations to transform applications into their distributed form. Still, both tools require modifications to the source code of the user application to insert parallel-specific code prior to actually Grid-enabling their compiled counterpart.

3 THE BYTECODE GRIDIFIER

To address the above problems, we propose BYG (BYtecode Gridifier) Mateos et al. (2009b), a general-purpose gridification tool that allows developers gridifying their ap-

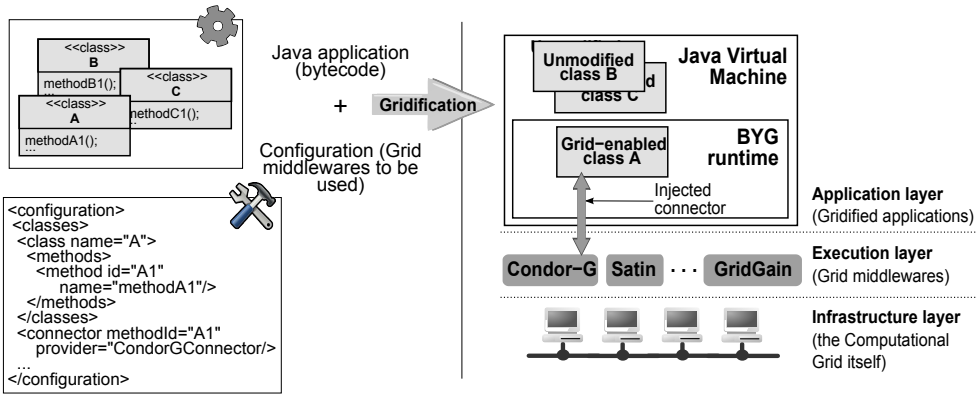

Figure 1: Overview of BYG

plications while requiring a minimal effort. To this end, BYG removes the need for explicitly alter the 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 application bytecode to delegate the execution of certain parts of the application to a Grid platform. The mapping of which parts are Grid-enabled is specified by means of user-supplied configuration external to the application being gridified.

Figure 1 depicts an overview of BYG. Conceptually, our approach takes as input the bytecode of an ordinary Java application, and dynamically transforms some of 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 this case method1), and delegates their execution to the target middleware (in this case Condor-G Thain et al. (2003)) 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, this is, 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 operation 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. parameters) 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 requeriments) must be addressed 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 program in the command that initiates the execution of the user application. BYG provides a connector for accessing the services of Satin Wrzesinska et al. (2006), a Java-based framework for parallelizing applications on LANs and WANs[1]. However, the development of connectors for other Grid middlewares is underway. This will enable developers to take advantage of features not present in Satin such as monitoring of running computations or data access. The next subsection explains how to exploit BYG from a practical perspective.

3.1 Gridifying applications with the BYtecode Gridifier

To gridify a conventional application with BYG, it is necessary to supply a configuration file (XML[2] format), which lists both the application classes to be Grid-enabled and the Grid 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 machine 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 machine, 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 Thain et al. (2003) and Globus Foster (2006) platforms, respectively. In summary, 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>

[1]The prototype is available for download at http://www.exa.unicen.edu.ar/~cmateos/projects.html
[2]A brief tutorial on XML can be found at http://www.w3schools.com/xml

element. Connectors are implemented through different classes that are shipped together with the BYG runtime.

3. For each one of the connectors, the IP and the port of the Grid machine 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 Vaughan-Nichols (2002). The IP, 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 somepackage.AdaptiveIntegration 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="somepackage.AdaptiveIntegration">
      <methods>
        <method id="mymethod" name="integrate">
          <parameter name="a" type="double"/>
          <parameter name="b" type="double"/>
          <parameter name="epsilon" type="double"/>
        </method>
      </methods>
    </class>
  </classes>
  <!-- Connectors to use -->
  <connectors>
    <connector methodId="mymethod" bindingId="mybinding"
            provider="org.isistan.byg.connectors.CondorGConnector"/>
  </connectors>
  <!-- Middleware-specific bindings -->
  <bindings>
    <binding id="mybinding" name="condor">
      <property name="protocol">sockets</property>
      <property name="address">condor_manager_ip</property>
      <property name="port">condor_manager_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 (a single connector can be used from different methods), and finally 1:1

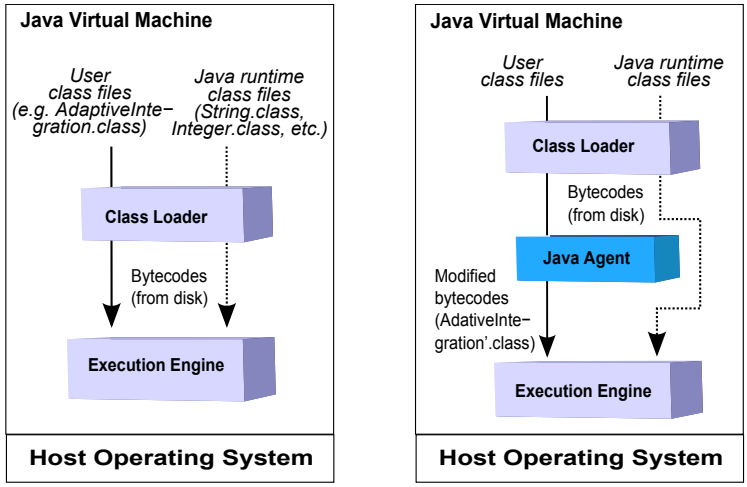

Figure 2: Modifying user classes on the fly: Java agents

relationships between connectors and bindings. In the example, we 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 employing socket-based communication. This bridging is performed by the CondorGConnector class from the BYG library, and the BYG core runtime, which transparently injects this class into the compiled code of the AdaptiveIntegration application class so that 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 Java Virtual Machine contacts (if defined) the corresponding Java agent and loads the bytecode resulted from passing the class through the agent. Figure 2 shows the differences between running a Java application in the usual way, this is, 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 can intercept 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 the 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 active the BYG agent), the startup command that launches the application must look like:

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

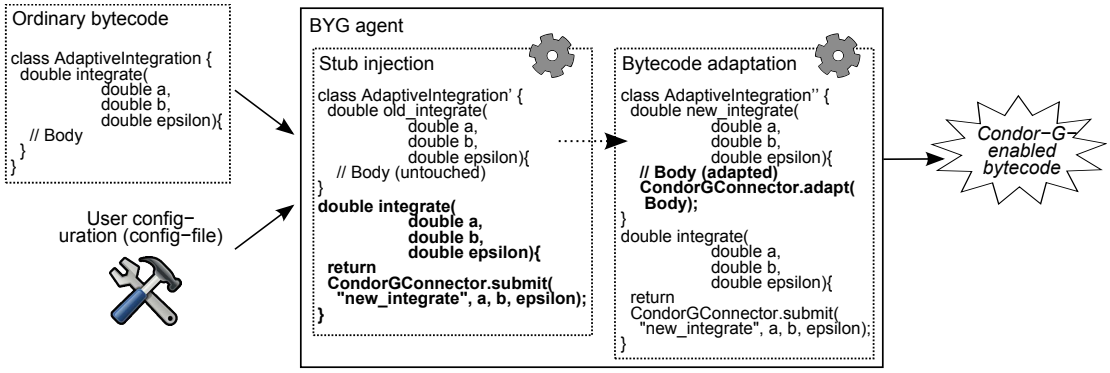

Figure 3: Overview of the BYG agent

The -javaagent switch instructs the Java Virtual Machine to use the Java agent implemented by the byg.jar user library. The characters enclosed within the "<" and ">" are the options for initializing the agent library. 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 Java Virtual Machine. To this end, BYG employs ASM ObjectWeb Consortium (2009), 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 transparently 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. Figure 3 depicts an overview of the mechanism implemented by the BYG agent to dynamically 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 Mateos et al. (2009b). 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 Wrzesinska et al. (2006) is a Java framework for programming parallel divide and conquer applications on local-area and wide-area clusters. Satin provides programmatic mechanisms for indicating which methods of a sequential application are parallelized and supplying synchronization for 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 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. Subsection 3.2.2 presents an overview of our Satin connector.

3.2.1 Satin: Programming model

Divide and conquer 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. Divide and conquer algorithms are usually implemented recursively, this is, 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.

For example, 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. 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) + function((b-a)/2+a));
7      double right = (delta/2) * (function(b) + function((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, this is, the case when the problem at hand becomes small enough to be solved directly.

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. Satin provides 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 an independent parallel subtask. However, we have to explicitly indicate in the application code the points in which it is necessary to wait for the child computations to complete. This is like providing a "join" point or barrier that causes the current task not to proceed and to wait for the 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 sentences 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 sentences and more complex control structures, it can be tedious and 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

The Satin connector 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 components, BYG relies on a software layer that wraps the Satin runtime. For more details on this extended Satin runtime, see Mateos et al. (2009b).

Besides injecting instructions to transparently execute ordinary methods on Satin (the "Stub injection" task in Figure 3), 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 automatically 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 sequen-

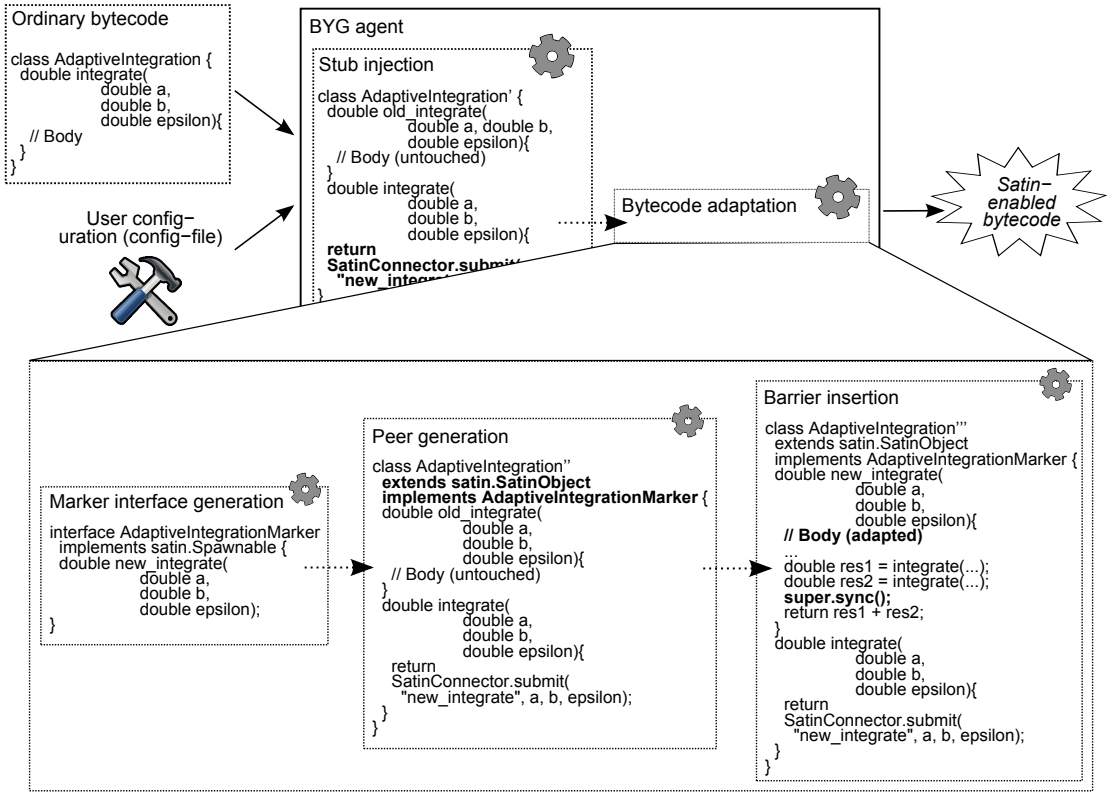

Figure 4: Satin-enabling ordinary bytecode: the Satin connector

tial 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 previous peer. The heuristic aims at preserving the operational semantics of the (sequential) original algorithm while minimizing the calls to the primitive.

Figure 4 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 automatically 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 transparently 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.SatinConnector"/>
  </connectors>
  <!-- Middleware-specific bindings -->
  <bindings>
    <binding id="mybinding" name="satin">
      <property name="protocol">sockets</property>
      <property name="address">satin_server_ip</property>
      <property name="port">satin_server_port</property>
    </binding>
  </bindings>
</configuration>
```

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 sentence. 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 method sentences 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. It is out of the scope of this paper to discuss the internals of this heuristic algorithm. For details on it, please refer to Mateos et al. (2009b).

4 EXPERIMENTAL EVALUATION

In Mateos et al. (2009b), we reported the performance of BYG by experimenting with it on a small LAN. To better evaluate BYG, we compared the performance that resulted from using Satin versus the BYG/Satin connector by running seven classic divide and conquer applications on another local cluster and a simulated wide-area Grid, namely:

- Prime factorization (PF): Splits an integer I into its prime factors. The multiplication of these factors is equal to I. For the experiments, we used $I = 15,576,890,767$.
- The set covering problem (Cov): Finds a minimal number of subsets from a list of sets L which covers all elements within L. The problem takes as a parameter the size of L. We used a list L that contained 33 sets with random elements.
- The knapsack problem (KS): Finds a set of items, each with a weight W and a

value V, so that the total value is maximal, while not exceeding a fixed weight limit. The problem receives as a parameter the initial number of items. We used a set of 33 items with random weights and values.

- Fast Fourier transform (FFT): Approximates a continuous function by a sum of sinusoids by using a finite number of points P sampled over a regular interval. We used $P = 2,097,152$.

- Adaptive numerical integration (Ad): Approximates the integral of a function $f(x)$ within a given interval (a, b) by replacing its curve by a straight line from $(a, f(a))$ to $(b, f(b))$, and then computing the area bounded by the resulting trapezoid. This is done by recursively further dividing the input interval into two subintervals (a_1, b_1) while the area of the trapezoid differs in a significant way from the sum of the areas of the trapezoids of the subintervals. The application receives as parameters $f(x)$, a, b and an $epsilon$ that establishes an upper bound for such difference. We used $f(x) = 0.1 * x * sin(x)$, $a = 0$, $b = 250,000$ and $epsilon = 0.000001$.

- Fibonacci series (Fib): Computes the Fibonacci number for some integer I by taking advantage of the recursive nature of the Fibonacci function. We used $I = 42$.

- Matrix multiplication (MM): Implements the popular Strassen's divide and conquer algorithm for matrix multiplication. We used a matrix of $3,072$x$3,072$ with random cell values.

Basically, "PF", "Cov" and "KS" are in essence NP problems, whereas the rest of the applications are benchmarks commonly employed to evaluate Grid frameworks. Due to their diversity in terms of functionality and resource demands, the applications used in our experiments were representative and thus provided the basis for a significant evaluation. For the sake of fairness, the source codes of the applications were obtained from the Satin project[3]. The variants for feeding BYG were obtained by removing from the original Satin codes any sentence related to parallelism and/or tuning application execution to derive the sequential divide and conquer counterparts of the applications.

First, we set up a cluster composed of 15 machines running Mandriva Linux 2009.0, Java 5 and Satin 2.1 connected through a 100 Mbps LAN. We used 8 single core machines with a 2.80 MHz CPU and 1.25 MB of RAM, and 7 single core machines with a 3 MHz CPU and 1.5 MB of RAM. Figure 5 shows the average execution time for 25 runs of these applications. In all cases, deviations were below 5%. Despite being an acceptable deviation when experimenting on wide area Grids, note that this percentage is rather high for a LAN-based cluster. The cause of this effect is that Satin –and thus the BYG/Satin and pure Satin applications– relies on a random task stealing algorithms.

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

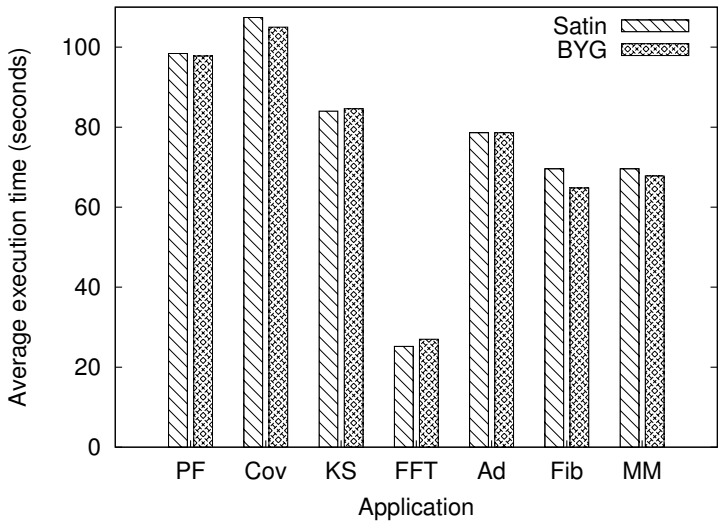

Figure 5: Performance of the test applications in the local cluster

All in all, BYG performed in a very competitive way compared to Satin, in spite of the fact that BYG adds some technological noise, which intuitively should translate into performance overheads. Basically, this noise is caused by the extended Satin runtime that handles the execution of peers in parallel. In fact, it can be seen that for 4 out of 7 test applications ("PF", "Cov", "Fib", "MM") BYG introduced performance gains with respect to Satin, which are explained by the differences between 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/Satin counterparts were parallelized by applying our heuristic on the sequential recursive codes derived from the pure Satin applications. However, our utmost goal is not to outperform existing parallel libraries like Satin, but simplifying their usage without incurring in an excessive penalty in terms of performance. This experiment shows that the BYG/Satin connector eases the construction of parallel applications, while stays competitive compared to directly employing Satin, which is explained by the effectiveness of our generic heuristic. Similar experiments confirming this result but performed on a more heterogeneous LAN can be found in Mateos et al. (2009b).

Later, we executed a subset of the above applications on a wide-area cluster, which was established by using WANem TATA Consultancy Services (2008), a software for simulating WAN conditions over a LAN environment. We simulated 3 Internet-connected local clusters C_1, C_2 and C_3 by using 4, 5 and 6 of the machines of the local cluster, respectively. Each WAN link was a T1 connection (i.e. a bandwidth of 1,544 Mbps) with a round-trip latency of 200 ms and a jitter of 10 ms, therefore the latency was in the range of 190-210 ms. Both the BYG/Satin and the pure Satin variants of the appli-

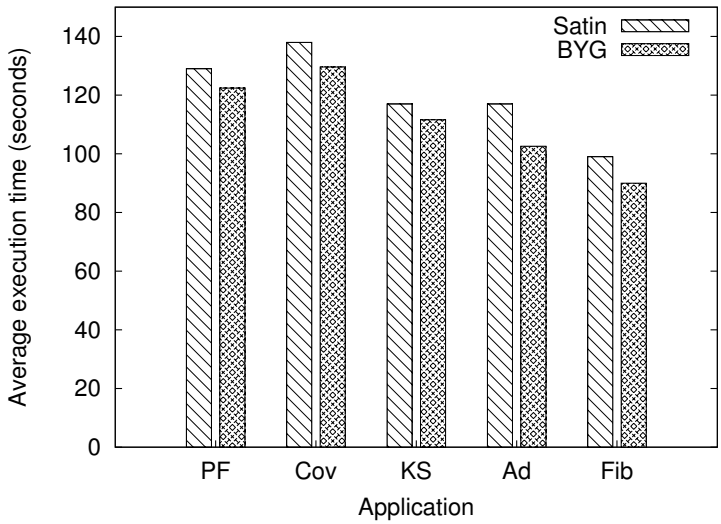

Figure 6: Performance of the test applications in the wide-area cluster

cations were configured to use the Cluster-aware Random Stealing (CRS) Wrzesinska et al. (2006) algorithm provided by the scheduler of Satin, instead of its default Random Stealing algorithm. With CRS, when a machine becomes idle, it attempts to steal an unfinished task from both remote or local machines, but intra-cluster steals have a greater priority than wide-area steals, which saves bandwidth and minimizes WAN latencies. Furthermore, the computation to network data transfer ratio of "FFT" and "MM" in this setting was very small, which severely and negatively affected processor usage. Therefore, we decided to left both applications out of the analysis since they did not experienced a CPU-intensive behavior in this testbed.

Figure 6 depicts the average execution time for 40 runs of the selected applications. In all cases, deviations were around 11%, which as explained before is mainly due to the random nature of the Satin scheduler plus the fact that we used WAN links to connect the three clusters. As shown, the BYG applications performed better than their respective Satin versions. Similar effects were observed with our extended Satin runtime in real wide-area Grids Mateos et al. (2009a). For the case of "KS" and "Ad", and unlike the previous LAN experiment, BYG outperformed Satin. Moreover, for the case of "PF", "Cov" and "Fib", the performance gains introduced by BYG were even greater than the gains obtained for these applications in the LAN setting (1, 2 and 7% versus 5, 6 and 9%, respectively). Hence, running these five applications in the wide-area cluster accentuated the differences regarding the way synchronization is added when using both tools, which lead to better performance in favor of the BYG applications. However, this trend should be further corroborated. All in all, these results show that the BYG applications performed very well, which aligns with the promissory results obtained in the experiments performed in the LAN environment.

5 CONCLUSIONS AND FUTURE DIRECTIONS

In this article, we presented BYG, an new tool to easily porting conventional compiled Java applications to Computational Grids. Its goal is to let developers gridify the binary code of existing applications and at the same to select which parts of the compiled code should run on a Grid. BYG targets Java applications implemented under the well-known 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 several computing intensive applications via BYG as well as Satin on a local and a wide-area Grid. Most of the BYG versions performed similarly to their Satin counterparts. We believe this is an interesting result considering that, to gridify an application, only some configuration have to be provided. However, we will conduct experiments with other applications and other Grid settings. For example, we are working on the gridification of a sequence alignment application on a large real (not simulated) wide-area Grid. Experiments will be performed based on real sequence databases extracted from the National Center for Biotechnology Information (NCBI) Web site[4]. In addition, we are extending the Satin connector to recognize more high-level Java sentences (e.g. try/catch) and to optimize the insertion of Satin barriers. Finally, we are investigating how to generalize and implement our gridification technique for other Grid frameworks besides Satin, and even other widely-adopted languages for scientific computing, such as C and C++.

REFERENCES

Baduel L., Baude F., Caromel D., Contes A., Huet F., Morel M., and Quilici R. *Grid Computing: Software Environments and Tools*, chapter Programming, Composing, Deploying on the Grid, pages 205–229. Springer, Berlin, Heidelberg, and New York, 2006. ISBN 1-85233-998-5.

Baitsch M., Li N., and Hartmann D. A toolkit for efficient numerical applications in Java. *Advances in Engineering Software*, 2009. ISSN 0965-9978. To appear.

Drummond L.A., Galiano V., Migallón V., and Penadés J. Interfaces for parallel numerical linear algebra libraries in high level languages. *Advances in Engineering Software*, 40(8):652–658, 2009. ISSN 0965-9978.

Eyheramendy D. *Innovation in Engineering Computational Technology*, chapter High abstraction level frameworks for the next decade in computational mechanics, pages 41–61. Saxe-Coburg Publications, 2006. ISBN 1-874672-28-8.

Foster I. The Grid: Computing without bounds. *Scientific American*, 288(4):78–85, 2003. ISSN 0036-8733.

Foster I. Globus toolkit version 4: Software for service-oriented systems. *Journal of*

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

Computer Science and Technology, 21(4):513–520, 2006. ISSN 1000-9000.

Hernández E., Cardinale Y., and Pereira W. Extended mpiJava for distributed checkpointing and recovery. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, volume 4192 of *Lecture Notes in Computer Science*, pages 158–165. Springer, Berlin / Heidelberg, 2006. ISBN 978-3-540-39110-4. ISSN 0302-9743.

Jugravu A. and Fahringer T. JavaSymphony, a programming model for the Grid. *Future Generation Computer Systems*, 21(1):239–246, 2005. ISSN 0167-739X.

Laskowski E., Tudruja M., Olejnik R., and Toursel B. Byte-code scheduling of Java programs with branches for Desktop Grid. *Future Generation Computer Systems*, 23(8):977–982, 2007. ISSN 0167-739X.

Lee E.A. The problem with threads. *Computer*, 39(5):33–42, 2006. ISSN 0018-9162.

Mackie R.I. Design and deployment of distributed numerical applications using .NET and component oriented programming. *Advances in Engineering Software*, 40(8):665–674, 2009. ISSN 0965-9978.

Manolescu D., Beckman B., and Livshits B. Volta: Developing distributed applications by recompiling. *IEEE Software*, 25(5):53–59, 2008. ISSN 0740-7459.

Mateos C., Zunino A., and Campo M. A survey on approaches to gridification. *Software: Practice and Experience*, 38(5):523–556, 2008. ISSN 0038-0644.

Mateos C., Zunino A., and Campo M. Grid-enabling applications with JGRIM. *International Journal of Grid and High Performance Computing*, 1(3):52–72, 2009a. ISSN 1938-0259.

Mateos C., Zunino A., Campo M., and Trachsel R. *Parallel Programming, Models and Applications in Grid and P2P Systems*, chapter BYG: An Approach to Just-in-Time Gridification of Conventional Java Applications. Advances in Parallel Computing. IOS Press, Amsterdam, The Netherlands, 2009b. ISBN 978-1-607-50004-9.

National Science Foundation. ScaLAPACK. http://www.netlib.org/scalapack (last accessed July 2009), 2007.

ObjectWeb Consortium. ASM. http://asm.objectweb.org (last accessed July 2009), 2009.

Papadimitriou S. and Terzidis K. jLab: Integrating a scripting interpreter with Java technology for flexible and efficient scientific computation. *Computer Languages Systems and Structures*, 35(3):217–240, 2009. ISSN 1477-8424.

Ropo M., Westerholm J., and Dongarra J. *Recent advances in Parallel Virtual Machine and Message Passing Interface - Proceedings of the 16th European PVM/MPI Users' Group Meeting, Espoo, Finland, September 7-10, 2009*. Lecture Notes in Computer Science. Springer-Verlag, Berlin / Heidelberg, 2009. ISBN 978-3-642-03769-6.

Seymour K. and Dongarra J. Automatic translation of Fortran to JVM bytecode. *Concurrency and Computation: Practice and Experience*, 15(3-5):207–222, 2003. ISSN 1532-0626.

Shafi A., Carpenter B., and Baker M. Nested parallelism for multi-core HPC systems using Java. *Journal of Parallel and Distribed Computing*, 69(6):532–545, 2009a. ISSN 0743-7315.

Shafi A., Carpenter B., Baker M., and Hussain A. A comparative study of Java and C performance in two large-scale parallel applications. *Concurrency and Computation: Practice and Experience*, 21(15):1882–1906, 2009b. ISSN 1532-0626.

TATA Consultancy Services. WANem. http://wanem.sourceforge.net (last accessed July 2009), 2008.

Thain D., Tannenbaum T., and Livny M. Condor and the Grid. In F. Berman, G. Fox, and A. Hey, editors, *Grid Computing: Making the Global Infrastructure a Reality*, pages 299–335. John Wiley & Sons, New York, NY, USA, 2003. ISBN 0-47085-319-0.

University of Virginia. jPVM. http://www.cs.virginia.edu/~ajf2j/jpvm.html (last accessed July 2009), 1999.

Vaughan-Nichols S.J. Web Services: Beyond the hype. *Computer*, 35(2):18–21, 2002. ISSN 0018-9162.

Wrzesinska G., van Nieuwport R., Maassen J., Kielmann T., and Bal H. Fault-tolerant scheduling of fine-grained tasks in Grid environments. *International Journal of High Performance Computing Applications*, 20(1):103–114, 2006. ISSN 1094-3420.

Zhang B.Y., Yang G.W., and Zheng W.M. JCluster: An efficient Java parallel environment on a large-scale heterogeneous cluster. *Concurrency and Computation: Practice Experience*, 18(12):1541–1557, 2006. ISSN 1532-0626.

Zhang H., Lee J., and Guha R. VCluster: A thread-based Java middleware for SMP and heterogeneous clusters with thread migration support. *Software: Practice and Experience*, 38(10):1049–1071, 2008. ISSN 0038-0644.