Advances in Engineering Software 43 (2012) 27–43

Contents lists available at SciVerse ScienceDirect

Advances in Engineering Software

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

Enhancing the BYG gridification tool with state-of-the-art Grid scheduling mechanisms and explicit tuning support

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

[a] ISISTAN Research Institute, UNICEN University, Campus Universitario, Tandil (B7001BBO), Buenos Aires, Argentina
[b] Consejo Nacional de Investigaciones Científicas y Técnicas (CONICET), Argentina
[c] UNICEN University, Campus Universitario, Tandil (B7001BBO), Buenos Aires, Argentina

ARTICLE INFO

Article history:
Received 25 June 2011
Received in revised form 25 August 2011
Accepted 26 August 2011
Available online 25 September 2011

Keywords:
Grid computing
Gridification
Java
BYG
Policies
Satin
GridGain

ABSTRACT

Grid Computing allows scientists and engineers to run compute intensive experiments that were unfeasible not so long ago. On the downside, for users not proficient in distributed technologies, programming for Grids is difficult, tedious, time-consuming and error-prone. Then, disciplinary users typically waste precious time that could be instead invested into analyzing results. In a previous paper, we introduced BYG (Mateos et al., 2011) [28], a Java-based software that automatically parallelizes sequential applications by directly modifying their compiled codes. In addition, BYG is designed to harness Grid resources by reusing existing Grid platforms and schedulers. In its current shape, however, BYG lacks support for some state-of-the-art Grid schedulers and mechanisms for introducing application-dependent optimizations to parallelized codes. In this paper, we present several extensions to BYG aimed at overcoming these problems and thus improving its applicability and delivered efficiency. We also report experiments by using traditional computational kernels and real-life applications to show the positive practical implications of the proposed extensions.

1. Introduction

Grid Computing [9] is a well-established distributed computing paradigm that allow scientists and engineers to build applications that demand by nature a huge amount of computational resources (e.g. CPU cycles, memory, disk). To this end, Grid Computing platforms provide the illusion of the existence of a large virtual supercomputer, which in turn virtualizes and combines the hardware capabilities of a number of much less powerful, geographically-dispersed machines. Common uses of Grid Computing environments include aerodynamic design, weather prediction, catastrophe simulation, financial modeling, and so on.

The inherent distributed and parallel nature of Grid applications, however, places a huge burden on regular users willing to exploit the hardware capabilities of such supercomputers, since a significant development effort and knowledge on distributed and parallel programming are required to put a Grid application to work [26]. Particularly, when deploying an algorithm to a Grid for execution, a user must take into account what his application does from a functional perspective as well as how to parallelize it according to the characteristics of the underlying Grid execution infrastructure. The second aspect is absent when developing traditional, single-machine applications, and is rather difficult to undertake by users with limited knowledge on Grid programming concepts.

The traditional approach to address the problem of simplifying Grid programming is based on supplying users with simple and designed-from-scratch programming APIs. In this arena, MPI [34] and PVM [34] appear as the most popular API-based programming tools among scientists and practitioners. Specifically, both tools offer intuitive (and standardized) library calls through which users can parallelize an application and execute parts of it in a distributed environment. However, even though MPI and PVM greatly mitigate the complexity inherent to writing Grid applications, their APIs still require users to have a solid knowledge in parallel and distributed programming [41]. Another problem of these tools is that they are essentially *intrusive*, i.e. parallelizing a sequential algorithm means "polluting" its code with a lot of parallel directives in the form of annotations and function calls. Then, there is not a clear separation between algorithmic code and parallel one. Consequently, introducing purely algorithmic optimizations remain error prone and require a considerable amount of testing [26], which demands time and effort.

The recent notion of "gridification" [26] has introduced a radical twist in the way Grid applications are built. Tools materializing this notion operate by automatically building a Grid-aware code from its sequential version. Amongst the advantages of this new approach compared to the traditional way of constructing Grid applications mentioned above are precisely supporting non-expert

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

0965-9978/$ - see front matter © 2011 Elsevier Ltd. All rights reserved.
doi:10.1016/j.advengsoft.2011.08.006

developers, avoiding the inclusion of API-specific parallel instructions within the source code of sequential applications, and in general simplifying and accelerating Grid application bootstrapping and execution. However, materializing the concept is challenging from both a conceptual and a technological standpoint [28]. Besides, much research is still being conducted to determine whether the performance achieved by automatically-gridified applications is competitive with that of hand-coded Grid-aware codes.

In a previous paper, we proposed BYG (BYtecode Gridifier) [28], a gridification tool that accepts as input the compiled code of a sequential application developed in Java and automatically outputs its parallelized counterpart. This process roughly comprises two broad tasks, namely heuristically detecting prospective portions within the input application for inserting parallelism, and including proper bytecode for actually executing the Grid-enabled binary code on a specific Grid platform. BYG targets Java applications developed under the divide and conquer model (D&C), a versatile technique for algorithm design by which an individual problem is solved by dividing it into several subproblems until trivial[1] subproblems are obtained. Upon executing a sequential D&C application, BYG modifies its bytecode so that subproblems are executed in parallel on a Grid via an existing task scheduler. Preliminary experiments have confirmed the feasibility of the approach.

Despite the encouraging results obtained so far, we believe that, in its current shape, the broad applicability of BYG is still somehow compromised. On one hand, to execute parallelized applications, BYG does not reinvent the wheel by providing yet another distributed task scheduler. Instead, a key design driver of its runtime was to exploit existing Grid platforms. However, at present, BYG is integrated with only one Grid platform. On the other hand, BYG is based exclusively on an implicit form of gridification by which parallelism is introduced transparently in an application, i.e. by requiring almost no user intervention. Clearly, this *implicit* form of parallelism allows users to gridify applications without thinking about parallelism. With the *explicit* approach to parallelism followed by tools such as MPI and PVM, or newer Grid platforms like Satin [40] and GridGain [13], the burden of managing parallelism falls on developers. However, explicit parallelism supplies APIs so that developers have more control over parallel programming, and thus potentially more efficient applications can be built [10].

Consequently, we have been working on addressing these issues, which resulted in a number of extensions to the software tool presented in [28]. In this sense, in this paper we introduce the following relevant contributions:

- The integration of BYG with more Grid middlewares, an hence the inception of newer bytecode rewriting mechanisms. This feature indirectly offers users a broader range of Grid schedulers to execute their gridified applications. The extended version of BYG offers binding to Satin [40],[2] a well-established and healthy academic project, and GridGain [13],[3] a relentlessly growing commercial platform for Grid Computing.
- A simple programming model based on *policies*, or rules that allows users to "throttle" the amount of parallelism and control task location in their applications according to both the nature of their codes and the characteristics of the Grid environment where they run. Rules can be specified in Java, Python and Groovy. This extension essentially aims at providing a balance to the simplicity versus performance tradeoff inherent to implicit and explicit parallel programming.

[1] Whether a problem is trivial or not depends on the nature of the application at hand.

[2] The full version is available at http://users.exa.unicen.edu.ar/cmateos/files/BYG-1.0.zip.

[3] A preliminary out-of-the-box version is available at http://code.google.com/p/easyfjp-imp.

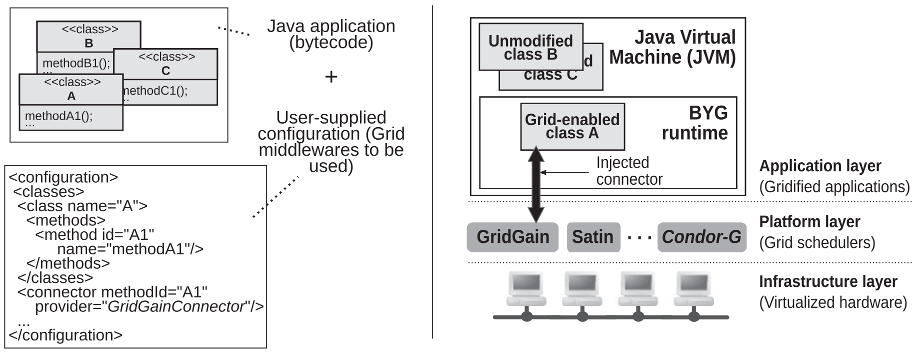
To evaluate the extended tool, we conducted two set of experiments. First, we executed several microbenchmarks to quantify the performance penalty of supporting Java-based and script-based policies. Second, we assessed the effectiveness of our extensions in terms of execution performance by using some explicit parallel models, BYG and policies to gridify two resource-intensive applications, namely the ray tracing 3D scene rendering technique and an algorithm for pairwise gene sequence alignment, on an emulated Grid. Basically, we compared hand-coded applications using contemporary Grid programming libraries against codes also exploiting these libraries but automatically obtained by using BYG plus policies. We believe that the competitive performance levels achieved by BYG across the various experimental scenarios and the improved flexibility and applicability of the extended gridification mechanisms, make BYG a valuable alternative for rapidly executing both engineering and scientific applications while efficiently exploiting Grid resources.

The rest of the paper is structured as follows. Section 2 overviews BYG by taking a user-centric approach to describe its capabilities in terms of application programming and integration with external Grid platforms. After that, Section 3 explains the extensions for tuning BYG applications and provides several source code examples. Section 4 reports a detailed experimental evaluation of BYG. Later, Section 5 discusses relevant related efforts. Finally, Section 6 concludes the paper.

2. The BYG (BYtecode Gridifier)

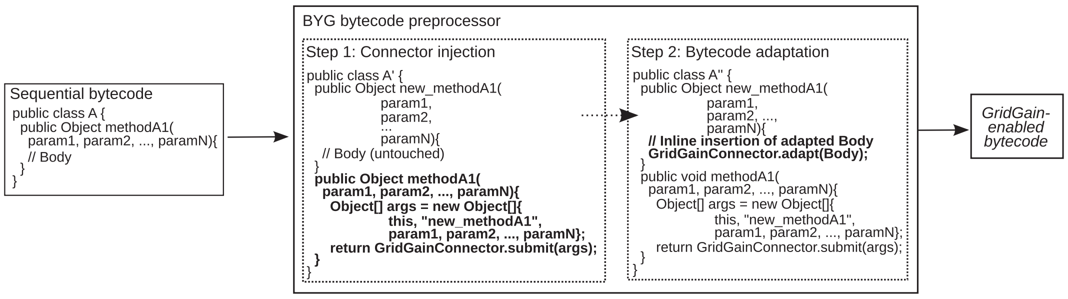
BYG is a gridification tool that allows developers to non-invasively gridifying their applications, or in other words, without tying their sequential codes to specific parallel and distributed libraries. Central to the gridification model promoted by BYG is the concept of Fork-Join Parallelism (FJP), a simple but effective technique that expresses parallelism via two primitives: *fork*, which starts the execution of a method in parallel, and *join*, which blocks a caller until the execution of methods finishes. FJP represents an alternative to thread-based parallel programming models, which have been criticized due to their inherent complexity [23], and the bureaucratic approach to parallelism of parallel libraries such as MPI [34] or PVM [34]. In fact, the Java language, which has offered threads as first-class citizens for years, includes now an FJP package for multicore CPUs (http://openjdk.java.net/projects/jdk7/features).

Certainly, FJP is not circumscribed to multicore programming, but is also applicable in execution environments where the notions of "task" and "processor" exist. Interestingly, multicore CPUs, clusters and Grids alike can execute FJP tasks, as they conceptually comprise processing nodes (cores or individual machines) interconnected through communication "links" (a system bus, a high-speed LAN or a WAN). This uniformity arguably allows the same FJP application to be run in either environments, provided there is a platform aware of the underlying execution support. Intuitively, FJP is suitable for parallelizing divide and conquer (D&C) applications, an algorithmic abstraction useful to solve many problems. This is because D&C applications are mostly developed in a recursive way, thus recursive calls can be mapped to independent parallel tasks. By basing upon these premises, BYG essentially contributes with bytecode analysis and generation techniques that automatically introduce FJP-based parallelism into sequential D&C applications. Broadly speaking, while most of the existing work on automatic parallelization of



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

Fig. 1. An overview of BYG: both the static (left) and runtime views (right) of a BYG application are illustrated.


Sequential bytecode
BYG bytecode preprocessor
Step 1: Connector injection
Step 2: Bytecode adaptation
GridGain-
enabled
bytecode

```
public class A {
  public Object methodA1(
    param1, param2, ..., paramN){
    // Body
  }
}
```

```
public class A' {
  public Object new_methodA1(
                 param1,
                 param2,
                 ...
                 paramN){
    // Body (untouched)
  }
  public Object methodA1(
    param1, param2, ..., paramN){
      Object[] args = new Object[]{
                this, "new_methodA1",
                param1, param2, ..., paramN};
      return GridGainConnector.submit(args);
  }
}
```

```
public class A'' {
  public Object new_methodA1(
                 param1,
                 param2, ...,
                 paramN){
    // Inline insertion of adapted Body
    GridGainConnector.adapt(Body);
  }
  public void methodA1(
    param1, param2, ..., paramN){
      Object[] args = new Object[]{
                this, "new_methodA1",
                param1, param2, ..., paramN};
      return GridGainConnector.submit(args);
  }
}
```

Fig. 2. Adapting sequential bytecodes to run on a Grid platform.

sequential codes focuses on loops [17], supporting D&C applications is interesting since there is an important class of algorithms that cannot be straightforwardly and efficiently expressed as loops because recursion is needed [1].

Another distinctive aspect of BYG is that, instead of providing its own task scheduler, our tool is designed to take advantage of the scheduling services of existing Grid platforms for executing parallelized applications. Architectonically, this is done through the use of *connectors*, which implement the necessary plumbing operations to access the execution services of specific Grid platforms. Apart from the code for supporting method-level parallelism itself, connectors are also non-invasively injected into the input sequential bytecode to delegate the execution of certain application methods to a Grid platform.

Fig. 1 depicts an overview of BYG. Our approach conceptually takes as input the executable code – i.e. the bytecode – of a sequential Java application, and dynamically transforms their classes to run on a Grid middleware the specific methods of the user's choice. 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, methodA1), and delegates their execution to the target middleware (in the example, GridGain) 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 running unmodified at the client side. BYG-enabling an application only requires the user to specify an XML file listing which methods of an application are to be gridified and what Grid platform must be employed to execute them. It is also possible to delegate different methods of classes belonging to the same application to various Grid middlewares.

From a runtime perspective, preparing or modifying an individual method for Grid execution involves two tasks. First, its body is rewritten to transparently delegate its execution to the connector associated to the method. In our case, a GridGain parallel job is created from the binary code of the target method, which is submitted to a Grid running GridGain. In this way, every time this method is invoked from the application, the method is not executed locally but an adapted version of it is handled by GridGain. The second task is precisely responsible for performing this adaptation, which is done by modifying the original bytecode of the method in order to exploit the parallel API library functions of the target platform. Depending on the middleware to be used for method execution, additional modifications at the class level may be necessary, as some platforms require jobs for example to extend from certain API classes. Fig. 2 overviews the mechanism described above, which dynamically obtains the gridified counterpart of a sequential class. Middleware-dependent transformations are the ones performed at step 2.

At present, BYG provides a connector for accessing the services of Satin [40], a Java-based platform for parallelizing and running applications on clusters and wide-area Grids. Also, we have developed a connector for the GridGain middleware [13]. Efforts towards providing a connector for the popular Condor-G platform [37] are also underway, whose implementation relies on a Java interface to this platform [30]. However, in the following subsections we will concentrate on Satin and GridGain as they are the most stable versions of our BYG connectors. Moreover, more details on the configuration needed to Grid-enable conventional applications and the core mechanics of the BYG runtime support can be found in [28]. In the following section we will focus on illustrating the mechanisms used by BYG to Grid-enable sequential applications from a user's perspective.

2.1. BYG: programming model

The divide and conquer (D&C) model is an algorithm construction technique that allows users to solve a problem by breaking it down into several subproblems of the same type until trivial problems are obtained. The solutions to the different subproblems are then combined to build the solution to the whole problem. Most of the time, D&C algorithms are implemented recursively, i.e. by issuing several recursive calls to the same code (method or function) implementing the problem. Moreover, trivial problems are processed directly, in the sense that they are not further subdivided but instead computed by a portion of non-recursive code.

Let us take for example the code shown below, which includes a recursive integrate method that computes the integral of a fixed function within a given interval (a, b). The integral value is approximated by systematically 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 associated to the subintervals is not smaller than some threshold epsilon:

```
1  public class AdaptiveIntegration {
2    public double function (double value){...}
3    public double integrate (double a, double b, double epsilon){
4      double delta=(b-a)/2;
5      double total = delta*(function (a)+function (b));
6      double deltaHalf = delta/2;
7      double left = deltaHalf*(function (a)+function (a + delta));
8      double right = deltaHalf*(function (b)+function (a + delta));
9      double diff = total-(left + right);
10     diff=(diff < 0)? -diff: diff;
11     if (diff < epsilon)
12       return total;// base case for D&C
13     else {
14       double res1 = integrate (a, a + delta, epsilon); // subproblem#1
15       double res2 = integrate (a + delta, b, epsilon); // subproblem#2
16       return res1 + res2;
17     }
18   }
19 }
```

The recursive calls of lines 14 and 15 are the *divide* phase of the algorithm, in which the problem at hand is large enough to be subdivided. Lines 11–12 represent its *conquer* phase, or the case when a (sub) problem becomes trivial. As the reader can see, for clarity purposes, the above code is not completely optimized since we have deliberately repeated some of the intermediate calculations.

2.2. Using BYG in conjunction with Satin

The Satin Grid platform refines the ordinary semantics of sequential D&C applications such as AdaptiveIntegration to introduce parallelism in the divide phase. The Satin library cleanly supports FJP through the *spawn* and *sync* primitives. The former allows users to create parallel subcomputations. Methods considered for parallel execution must be included in a regular Java interface (*marker interface*) that extend the satin.Spawnable interface. Moreover, the *sync*primitive is shipped as a library call and is used to programmatically block the execution of a task until the execution of its child tasks finish. For example, the Satin version of the AdaptiveIntegration class is:

```
1  public interface AdaptiveIntegrationMarker extends satin.Spawnable{
2    public double integrate (double a, double b, double epsilon);
3  }
4  public class AdaptiveIntegration extends satin.SatinObject
5                    implements AdaptiveIntegrationMarker{
6    ...
7    public double integrate (double a, double b, double epsilon){
8      ...
9      else {
10       double res1 = integrate (a, a + delta, epsilon);
11       double res2 = integrate (a + delta, b, epsilon);
12       super.sync ();
13       return res1 + res2;
14     }
15   }
16 }
```

Basically, AdaptiveIntegrationMarker indicates Satin which methods of the application under analysis must be executed in parallel and as such trigger independent parallel subtasks at runtime. Methods not included in the marker interface are executed in a sequential way. Also, users have to explicitly indicate in the application code the points in which it is necessary to wait for child computations to complete, or in other words providing a join point to cause subtasks not to proceed and to wait for divide parts of the problem. To this end, the call to sync at line 12 ensures that the subresults computed by the subtasks spawned at lines 10 and 11 are instantiated to build a larger result at line 13. This simple synchronization mechanism is at the same time the main source of programming errors when employing Satin. A rule of thumb for correctly using the sync primitive and therefore avoid attempting to access not yet computed subresults involves checking that at least one call to the primitive is performed between the statements including recursive calls and those that access their results. For more complex algorithms, however, this analysis is rather tedious, time-consuming and, even more important, error-prone.

Once the application has been (re) written to exploit the Satin API, it is necessary to pass the compiled counterpart of the parallel source code to a special postprocessor. Conceptually, this support transparently further modifies the application in such a way that, at runtime, an independent or forked task is created for every invoked recursive call. Tasks are in turn managed by a Satin scheduler capable of sending/retrieving tasks to/from remote hosts to take advantage of distributed computing resources.

Our Satin connector automates the previous manual tasks from a compiled D&C application not explicitly coded to use the Satin API. To this end, the Satin connector generates the marker interface based on the configuration of the application (i.e. the one depicted in Fig. 1, left), and rewrites the bytecode of the class being gridified (i.e. the class A in Fig. 1) to extend/implement the necessary classes and interfaces so that the sequential class follows the parallel application structure prescribed by Satin. The connector also inserts calls to sync by deriving a high-level representation from the bytecode and analyzing the points where joins are needed.

In summary, the connector carries out three main tasks. First, as explained, Satin requires applications to include a marker interface, which lists the methods considered for parallel execution. The connector builds this interface from the methods listed in the ⟨classes⟩ section of the user-supplied configuration. Second, Satin codes must implement a marker interface and to extend from SatinObject. Then, a clone (or in BYG terminology a *peer*) of the sequential class under consideration is created by the Satin connector and modified to fulfill these requirements. Lastly, the connector inserts calls to the Satin sync primitive at appropriate places of the spawnable methods of the peer based on an heuristic algorithm. The algorithm preserves the operational semantics of the (sequential) original algorithm while minimizing the calls to the primitive.

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 finish, our algorithm uses an heuristic to keep the correctness of the program while minimizing the inserted calls to sync for the sake of efficiency. For simplicity, we have left the discussion of the internals of this heuristic algorithm out of the paper.

2.3. Using BYG in conjunction with GridGain

GridGain is a very stable open source Grid platform that has became popular for developing distributed applications. The tool supports several programming models for developing parallel applications. Particularly, GridGain natively supports the classical master-worker paradigm by means of an extension to the standard Java futures package. In a broad sense, a *future* is an abstraction that allows users to represent and manipulate an individual asynchronous computations. The package offers an API that exposes objects and methods to parallelize applications on multicore machines and hides programmers from many low-level details related to parallelism such as thread creation and coordination. Basically, GridGain extends the package by allowing such threads (or tasks) to cooperatively execute not only on multiple cores but also across many machines.

The weak point of manually using GridGain, however, is the bureaucratic nature of its master-worker API, which even though it uses nice parallel abstractions based on conventional Java futures still demands parallel and distributed programming concepts from developers. Again, this is a threat to adoption when it comes to engineers and scientists. GridGain also provides alternative parallel development models apart from master-worker that have simpler APIs, but which also require expertise.

In this sense, we have designed an algorithm for both inserting barriers and generating parallel applications that exploits the master-worker API for D&C applications. Regarding barrier insertion, the algorithm is materialized as an adaptation to that of associated to Satin, and as such looks for definitions and uses of parallel variables for incorporating parallelism. With respect to code generation, a challenging issue concerns parallel programming model adaptation. Supporting libraries in BYG already based on D&C such as Satin mostly requires source-to-source translation, i.e. recursive methods in the input application are forked in the output application via proper calls to the target library API. But, libraries relying on conventional execution models – e.g. master-worker or bag-of-tasks – in which there are not *hierarchical*relationships between parallel tasks, is not straightforward as BYG must somehow adapt the task structure of the input D&C application. Precisely, an example of such a library is GridGain.

Let us illustrate the code obtained when generating parallel applications based on the GridGain library (version 2.1.0 [4]). To this end, we will use as input the D&C code given by the sequential version of the AdaptiveIntegrationclass presented at the beginning of Section 2.1:

```
1  import org.gridgain.grid.Grid;
2  import org.gridgain.grid.GridFactory;
3  import
   org.gridgain.grid.kernal.executor.GridExecutorCallableTask;
4  import org.gridgain.grid.GridTaskFuture;
5
6  import java.util.concurrent.Callable;
7  import java.io.Serializable;
8
9  public class AdaptiveIntegration implements Serializable{
10   public double function (double value){...}
11   public double integrate (double a, double b, double
   epsilon){
12   return new_integrate (a, b, epsilon);
13   }
14   // The GridGain-enabled method
15   protected double new_integrate (double a, double b,
   double epsilon){
16     Grid grid = GridFactory.getGrid ();
17     GridExecutorCallableTask exec = new
   GridExecutorCallableTask ();
18       ...
19       else {
20       //subproblem #1
21       GridTaskFuture < double> res1future = grid.execute (
22       exec, new AdaptiveIntegrationTask (this, a, a + delta,
   epsilon);
23       // subproblem #2
24       GridTaskFuture < double> res2future = grid.execute (
25       exec, new AdaptiveIntegrationTask (this, a + delta, b,
   epsilon);
26       return res1future.get () + res2future.get ();
27     }
28   }
29 }
30 // Subcomputation
31 public class AdaptiveIntegrationTask implements Callable{
32   //Instance variable declaration
33   public AdaptiveIntegrationTask (AdaptiveIntegration
   application,
34                       double a, double b, double epsilon){
35     // Copy arguments into instance variables
36   }
37   public Serializable call (){
38     return this.application.integrate (a, b, epsilon);
39   }
40 }
```

As shown in the example, the generated class contains a wrapper method (lines 11–13) that invokes the actual automatically parallelized method (lines 15–28), whose code has been derived from the original integrate method but modified to include GridGain forks and joins (lines 20–25 and 26, respectively).

Moreover, instances of AdaptiveIntegrationTask carry out the subcomputations by calling AdaptiveIntegration.integrate (double,

[4] http://www.gridgain.com/javadoc.

double, double) on individual branches of the whole execution tree. Although not shown in the source code snippet, there are other modifications that are introduced into the generated code in order to keep track of important runtime information such as the depth of the execution tree. The additional modifications are essentially glue code for invoking *policies* by passing along context information. The next section discusses the policy subsystem of BYG.

3. Optimizing BYG applications: a policy-based programming model

In a broad sense, a policy is a mechanism that allows developers to express, separately from the application logic, customized strategies to achieve better performance [25]. In practice, a policy is materialized via a user-specified rule that governs the parallel behavior of an application. As mentioned earlier, BYG was extended with a policy-inspired tuning support that let developers to introduce common optimization heuristics without altering their applications.

Policies considered by BYG are *application-specific* or *environment-specific*. The former group represents tuning decisions that depend on the algorithmic nature of the applications being parallelized. On one hand, application-specific policies model the notions of threshold (see Section 3.1), memoization (see Section 3.2) and task placement (see Section 3.3). A threshold policy establishes a defined limit to the number of parallel tasks spawned at runtime for an application. Memoization policies, complementary, allow users to reuse task results in those cases in which subcomputations overlap. Lastly, task mapping policies represent a mechanism to customize the physical location of spawned tasks.

On the other hand, environment-specific policies are optimization rules that regulate the amount of parallelism according to the computing capabilities and to some extent the topology of the underlying environment. To make decisions, these policies use dynamic information provided by the BYG runtime environment (CPU and memory availability, network conditions, and so forth). For coding environment-specific policies, BYG exposes a well-defined interface to system metrics. To this end, a profiling module is provided, through which users are able to query for example for the *overall* CPU load or the amount of parallel runtime tasks under execution within a cluster. Then, a user may code for instance a policy to relate the amount of parallelism of an application as an inverse function of the average CPU availability. Users can nevertheless develop policies combining application-specific optimizations with environmental conditions. For example, the amount of memoized results for a memory-intensive application may be controlled by also taking into account the available memory in the executing cluster. The following subsections focus on explaining application-specific policies, which are the most intuitive and useful for non-experienced developers. For more details on environment-specific policies, please refer to [27].

We have also developed a support for application-specific policies implemented in scripting languages. Currently, we support Python and Groovy. Firstly, Python is a very popular interpreted language among scientists [31,33]. Secondly, the Groovy language has recently proved to be a cost-effective and feasible alternative for coding engineering applications [32]. Section 3.4 explains this support.

3.1. Threshold-based policies

Threshold policies are useful for avoiding parallelizing a (sub) computation more than needed and otherwise run it sequentially. For example, in the AdaptiveApplication class, we may want to limit the number of generated parallel tasks that are injected into the runtime system depending on the depth of the execution tree of the method at runtime. This decision is indicated to BYG by associating the following policy to the integrate method:

```
import byg.policy.Policy;
import byg.policy.ExecutionContext;
public class MyThresholdPolicy implements Policy{
  static final int THRESHOLD = 100;
  public boolean shouldFork (ExecutionContext ctx){
    // integrate (a, b, epsilon)
    double a=(double) ctx.getArgument (0);
    double b=(double) ctx.getArgument (1);
    return ((b-a)>THRESHOLD);
  }
}
```

The code implements the Policy interface from the BYG policy API and allows each execution of integrate to be forked provided the difference between the two x-axis coordinates (i.e.a andb) is aboveTHRESHOLD. In this way, less parallel tasks are injected into the Grid, thus avoiding unnecessary scheduling overheads. ExecutionContext provides operations to further introspect the execution of the application, in this case obtaining the values of method parameters. The policy uses ExecutionContext to access the value of the first and second arguments of each call to integrate via the getArgument API method. To attach any policy to an application, such as the above threshold policy to the AdaptiveIntegration class, users must supply the corresponding declaration in the configuration of the application.

3.2. Memoization policies

Memoization is a common optimization technique that is useful to gain efficiency by avoiding forking a subcomputation when the same or similar results have been already computed by another subcomputation. From a programmer's perspective, coding a memoization policy requires deciding whether to fork or not, and in the latter case to identify the particular result that should be reused. For instance, let us suppose we have a D&C code for computing the N^{th} Fibonacci number:

```
public class Fibonacci{
  public long fibonacci (int n){
    if (n==0)
      return 1;
    long f1 = fibonacci (n-1);
    long f2 = fibonacci (n-2);
    return f1 + f2;
  }
}
```

Then, parallelizing this code with BYG allows the two recursive calls to execute in parallel. Naturally, the same applies to the two subproblems generated from either calls, and so forth. However, the overlapping nature of the subcomputations – i.e. the same code is run against the same set of inputs many times – makes an opportunity for an optimization based on memoization policies. In this sense, we could for example provide the following policy:

```
import byg.policy.MemoizationPolicy;
import byg.policy.ExecutionContext;
public class MyMemoizationPolicy implements
  MemoizationPolicy{
  public boolean shouldFork (ExecutionContext ctx){
    long n=(Long) ctx.getArgument (0);// fibonacci (n)
    return (n%2== 0);
  }
  public String buildResultKey (ExecutionContext ctx){
    return String.valueOf (ctx.getArgument (0));
  }
}
```

The policy indicates BYG to fork and hence to ignore previously computed results if the argument of a call to fibonacci is even. Moreover, whenever shouldFork evaluates to false, BYG attempts to reuse the value from a result cache with the key as indicated by buildResultKey. However, if shouldFork evaluates to false but the key is invalid and leads to a cache miss, the sequential execution of fibonacci takes place. For storing results, BYG is based at present on a distributed caching support built around the memcache library [27].

Although illustrative in our example, memoization strategies like the one implemented by MyMemoizationPolicy, in which only a subset of previously calculated results are reused, are useful to minimize the negative effects of querying the cache. This is since depending on the number of independent executing tasks, and the number of configured replicas storing cache entries, there may be far more queries than cache servers able to solve the queries, thus affecting performance. Furthermore, another use of such strategy is in parallel optimization problems where actually forking a subproblem may yield a better solution than reusing a similar computed suboptimal result [2]. All in all, memoization policies are useful for cache-friendly applications.

3.3. Task placement policies

Task placement refers to the problem of assigning unfinished tasks to available executing nodes. Broadly, tasks are mapped to computing nodes on an off-line (i.e. statically) and a runtime (i.e. dynamically) fashion, respectively [21]. Precisely, tasks resulted from executing D&C applications belong to the second category, because the execution of an individual task may trigger the execution of N more. In BYG, the node in charge of executing a task is not determined by the user application but the underlying Grid scheduler selected upon configuring connectors. However, the hierarchical task structure of BYG applications indirectly determines task dependencies that, unless considered by the scheduler, may cause the resulting performance to be suboptimal. Therefore, the goal of these policies is to allow the user to control the placement of forked tasks by selectively ignoring some of the decisions taken by the underlying task scheduler.

Consider, for instance, an application that performs some recursive computation on a quadtree data structure, such as an algorithm for localized image processing. Every parallel task creates four more tasks, each in charge of processing a particular region of the data (see Fig. 3). If we execute this application on a Grid comprising clusters connected through wide-area links, Fig. 4 (left) depicts one possible task mapping, assuming that we launch the execution of our application at cluster C_1. Alternatively, Fig. 4 (right) depicts another task mapping, by which $task_{d+1_1}$ and $task_{d+1_2}$ have been explicitly forced to be located at cluster C_1 and the placement of the rest of the siblings tasks of $task_{d_1}$ has been delegated to the scheduler. Depending on the amount of data interchanged between $task_{d_1}$ and $task_{d+1_1}/task_{d+1_2}$, the semi-automatic

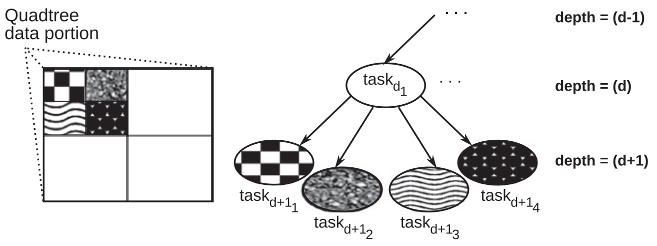


Fig. 3. Task placement example: an application processing a quadtree data structure.

mapping may greatly justify the loss of processing power available at cluster C_4.

Roughly, this decision can be specified through a task placement policy, which decides, based on an API object representing a task identifier, where to submit parallel tasks for execution:

```
import byg.policy.TaskPlacementPolicy;
import byg.policy.ExecutionContext;
import byg.policy.TaskId;
public class MyTaskPlacementPolicy implements
  TaskPlacementPolicy{
public boolean shouldMap (ExecutionContext ctx){
  //Avoid overloading the local node by checking whether
  //the current executing task's depth is odd or even
  if (ctx.getCurrentDepth ()% 2!= 0)
    return false;
  TaskId id = ctx.getCurrentTaskId ();
  //Only the first two quadrants (1 and 2) are explicitly mapped
  //Quadrants 3 and 4 are not affected by the policy
  return (id.getSubtaskNumber () < 3);
}
public String mapTo (ExecutionContext ctx){
    return "127.0.0.1";//Local IP address or hostname
  }
}
```

Basically, the shouldMap method tells BYG whether to activate explicit task mapping or not for a given subcomputation, whereas mapTo instructs the underlying platform to which node the task should be submitted. Each forked task is assigned a unique identifier that comprises its own identifier and a subtask identifier. In other words, if a given task whose identifier is I spawns N more tasks, spawned subcomputations are identified as I+"."+1, I+ "."+ 2, …, I+"."+N. For efficiency, identifiers are encoded as alphanumeric strings. In the example, we have forced the subtasks to be placed in the same physical node as the parent tasks originating them. However, other complex actions could had been taken, such as mapping tasks to any node of a given cluster, or even a cluster where a certain task is executing. Finally, task placement policies assume that the underlying Grid platform API has support for explicit task mapping, a feature that is not present in all Java-based parallel tools. However, many modern platforms such as GridGain [13] and ProActive [3] support this feature. Particularly, as BYG provides a connector for GridGain, users are allowed to employ task-mapping policies for their applications when using BYG in conjunction with this platform.

3.4. Script-based policies

Scripting languages enable rapid and easy development of applications and have become popular in scientific and engineering environments. One of the most attractive features of these lan-

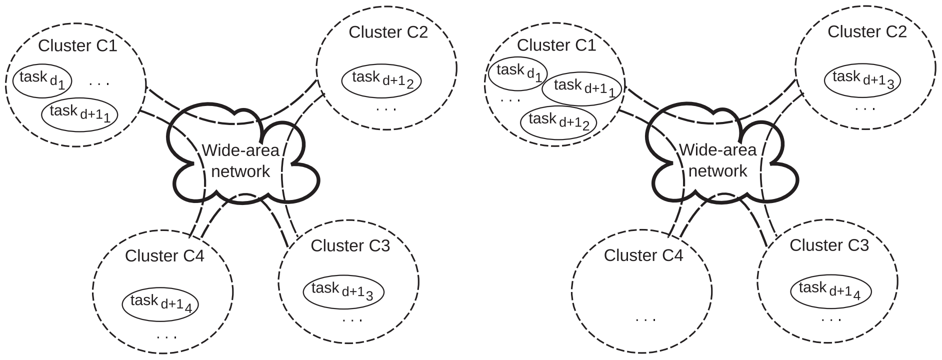


Fig. 4. Task placement example: scheduler-based (left) and semi-automatic (right) task mapping.

Table 1
Input and output objects available for coding policies.

Object type	Object	Default value	Description
Input	opCode	N/A. Its value can be one of "shouldFork", "buildResultKey", "shouldMap" or "mapTo"	Selector indicating the optimization function being executed
Output	ctx	N/A	Same as ExecutionContext for Java-based policies
	fork	*true*	Indicates whether to further split the current computation into subtasks or not
	map	*false*	Indicates whether explicit task mapping must be used or not
	resultKey	*null*	Holds the identifier associated to the task result to be reused
	destination	*null*	Holds the destination node to which the current computation must be placed

guages is faster application writing. In addition, most scripting languages are interpreted and as such can be exploited without the tedious code-compile-execute sequence. Furthermore, many scripting engines, nevertheless, allow scripts to be compiled to an intermediate representation that can be then executed. As a consequence, BYG offers support for executing threshold, memoization and task placement policies written in Python and Groovy. Basically, to use this support, users must provide an individual script file with.py or.groovy extension where the specific optimization to be used for an individual gridified method is declared. For efficiency reasons, policy scripts are compiled on background upon executing applications.

Irrespective of the scripting language employed, in order to introspect application execution and specify policy decisions, developers must manipulate some standard input and output objects defined by the BYG runtime. These objects are populated into the associated scripting engine upon evaluating a script. Input objects are read-only, whereas the value of output objects can be both read and modified. Table 1 lists the aforementioned objects. All objects have an untyped value field that actually contains the data associated to the object. Besides providing uniformity and thus simplifying object manipulation at the source code level, this allows BYG to abstract away datatypes differences across scripting languages. For example, boolean variables in some flavors of Python and Groovy are internally represented as numbers (0 is *false*) and booleans (*true* and *false*), respectively.

The following code shows an example policy that is basically the Python counterpart of the memoization policy discussed in Section 3.2:

```
def shouldFork ():
  fork.value = eq (ctx.value.getArgument (0) % 2, 0)
def buildResultKey ():
  resultKey.value = ctx.value.getArgument (0)
# Entry point of the script
if opcode.value=="shouldFork":
  shouldFork ()
elif opcode.value=="buildResultKey":
  buildResultKey ()
```

whereas the Groovy version of the task placement policy described in Section 3.3 is:

```
def shouldMap () {
  if (ctx.value.getCurrentDepth () % 2, 0) {
  def id = ctx.value.getCurrentTaskId ();
  if (id.getSubtaskNumber () < 3)
    map.value = true;
  }
}
def mapTo () {
  destination.value="127.0.0.1";
}
# Entry point of the script if (opcode.value=="shouldMap") {
  shouldMap ();
}
else
if(opcode.value=="mapTo") {
  mapTo ();
}
```

Of course, more complex policies than the ones exemplified above can be implemented. Furthermore, for developing this support, we followed a similar approach to the JLab [32] numerical computational environment, since we used the Java Scripting API[5] for managing the binding between Java and scripts. This API is shipped as a package bundled into the JVM since version 6 and offers a generic engine for executing scripts implemented in a variety of scripting languages (Python, Groovy, JavaScript, Rhino, Ruby, and many more).

4. Evaluation

This section describes the experiments that were performed to empirically evaluate the extensions to BYG described so far. We performed two type of experiments. On one hand, we executed several performance benchmarks to quantify the costs introduced by the extra source code for supporting Java-based policies inserted by BYG upon parallelizing user applications. This is reported in Section 4.1. Then, we evaluated the performance overheads of script-based policies. Roughly, the purpose of these experiments was to evaluate the effect of policies in parallelized applications without considering middleware-specific instructions. On the other hand, we assessed the effectiveness of BYG and policies to parallelize two real-life applications. These results are reported in Section 4.2. Comparisons were performed by using manually-generated parallel versions of their codes by using the GridGain parallel library, plus variants built by using the automatic parallelization support of BYG and policies.

4.1. Microbenchmarks

In order to quantify the execution overheads associated to BYG policies, we executed various benchmarks with our policy-based bindings to Java, Python and Groovy. Particularly, we used the following benchmark applications:

- *Ad* (adaptive numerical integration): Approximates a function $f(x)$ within a given interval (a,b) by replacing its curve by a straight line from $(a,f(a))$ to $(b,f(b))$. The application receives as parameters $f(x)$, a, b, and an *epsilon* that controls the mechanics of the algorithm. We used $f(x) = 0.1 * x * sin(x)$, $a = 0$, $b = 250000$. On the other hand, we used $epsilon = [0.001, 0.0001, 0.00001]$.
- *PF* (prime factorization): Splits an integer I into its prime factors such that their multiplication is equal to I. For the tests, we used $I = [155768907, 1557689076, 15576890767]$.
- *FFT* (Fast Fourier transform): Approximates a continuous function by a sum of sinusoids. The function is in turn approximated by a finite number of points P sampled over a regular interval. In the experiments, we used $P = [2097152, 4194304, 8388608]$.

As suggested, for *epsilon*, I and P we employed three different values, which in turn determined three input sizes – small, medium and large – for each application that increased the computational costs. Table 2 shows the number of recursive calls necessary to process the different inputs by the D&C sequential versions of the applications. On the other hand, the overheads introduced by the policy support in terms of extra Java objects allocated in RAM were left out of the analysis as they proved to be negligible.

To set the basis for comparison, for each application and input size, we executed four variants:

[5] http://java.sun.com/developer/technicalArticles/J2SE/Desktop/scripting.

Table 2
Recursive calls performed in the different runs.

Application/size	Small	Medium	Large
Ad	51,799,943	159,945,610	384,298,033
PF	32,767	557,054	4,751,357
FFT	4,194,303	12,582,910	29,360,125

Table 3
Java policies: overhead.

	Average runtime (s)			Overhead (s)	
	No policies	Threshold	Memoization	Threshold	Memoization
Small input					
Ad	14.569	14.673	16.169	0.104	1.600
PF	1.836	2.021	1.962	0.185	0.126
FFT	3.829	3.993	4.074	0.164	0.245
Medium input					
Ad	28.469	28.87	28.987	0.401	0.518
PF	18.322	18.682	19.068	0.360	0.746
FFT	7.23	7.638	7.367	0.408	0.137
Large input					
Ad	66.769	67.005	67.646	0.236	0.877
PF	183.386	186.461	187.302	3.075	3.916
FFT	15.432	16.308	15.552	0.876	0.120

1. A variant not relying on policies given by the original sequential D&C codes without BYG preprocessing.
2. A variant using a threshold policy that simply decides to fork upon every single call to shouldFork.
3. A variant using a memoization policy that forked execution provided the current depth is below a 75% of the final depth of the entire execution tree. The target depth in each case was determined beforehand and used to initialize the policy. For example, for the *Ad* application and input size *small*, this value was approximated by the formula $\rceil(0.75 * log_2 51{,}799{,}943)$. As a consequence, fewer forks compared to (2) were issued but cache usage was involved. Moreover, the buildResultKey method of the policy just returned a default value, therefore during execution spurious objects were put into the local result cache, which were retrieved but not reused.

Put simply, (2) was designed to stress out our basic policy runtime framework, while (3) represented a potentially common scenario of threshold-memoization combinations in order to provide an evaluation as realistic as possible. Lastly, for the sake of accuracy, we disabled the code insertions necessary to support task launching and synchronization, thus only the policy-related code from the BYG runtime was executed.

Table 3 shows the average execution time of the variants listed above for all input sizes. Experiments were executed on a Intel® Core i3 M380 CPU running at 2.53 GHz (only one core was used). It can be seen from the Table that the variants using policies incurred in some overheads, but in all cases they were below 4 seconds. On the other hand, in general, the overheads tended to decrease as the size of the experiments increased. For example, for small input sizes, policies added an execution overhead of up to 10%, whereas for large input sizes, the overhead was in the range of 1–5%. We could reasonably extrapolate these results to argue that, for very large input sizes and thus execution times in the order of hours, overheads would be negligible.

In a second round of tests, we run some experiments to evaluate the performance of script-based policies. Broadly speaking, interpreted languages are more expensive than compiled languages. Therefore, we decided to provide script-enabled versions of the

```
def shouldFork ():
 fork.value = 1
def buildResultKey ():
 resultKey.value = "foo"
# Entry point of the script
if opCode.value == "shouldFork":
 shouldFork()
elif opCode.value == "buildResultKey":
 buildResultKey()
```
	```
	def shouldFork () {
	 fork.value = true;
	}
	def buildResultKey () {
	 resultKey.value = "foo";
	}
	# Entry point of the script
	if (opCode.value == "shouldFork") {
	 shouldFork();
	}
	else
	if (opCode.value == "buildResultKey") {
	 buildResultKey();
	}
	```

Fig. 5. Python policy (left) and Groovy policy (right) used during the experiments.

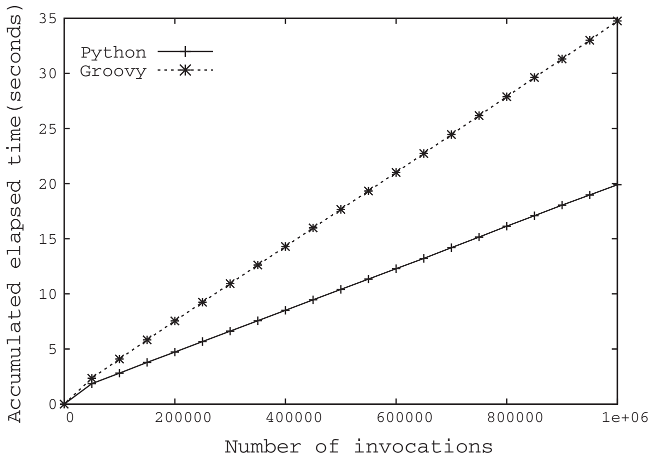


Fig. 6. Accumulated total elapsed time when executing the script-based policies.

benchmarks policies described before. We derived two policies, one based on Python and another one coded in Groovy (see Fig. 5). After that, we separately evaluated these policies up to 1,000,000 times by sequentially calling their shouldFork and buildResultKey methods. This allowed us to quantify the main cost component of supporting script-based policies since most of the associated performance penalty resides in the library classes of the JVM that are used to bind the Java code implementing these methods and their associated scripting codes. Results are shown in Fig. 6 by depicting the total elapsed time within accumulative windows of 1,000 runs. We also run the same experiment with the Java versions of the policy, however their associated elapsed times were not included in the graphic as they were very small.

As expected, evaluating script-based policies has associated an inherent cost that is explained by the interpreted nature of scripting languages. This does not mean that script-based policies are not viable, but rather they should be used sparingly by controlling the *granularity* of the computations. By granularity we refer to the number of parallel tasks in which a single unit of computation is split. In this way, the more the number of effective forks, the finer the granularity of the whole application (many tasks with small computational requirements). However, the cost of evaluating policies increases. Likewise, the less the number of forks, the coarser the granularity (few tasks with high computational requirements). Here, the negative incidence of policies in the execution time decreases. This clear trade-off can be formally expressed by the following formula:

$$Overhead = \frac{(2^{(N-1)} - 1) * cost_{policy}}{(2^{(N-1)} - 1) * cost_{policy} + (2^{N} - 1) * cost_{task}}$$

where:

- *Overhead* is the percentage of the total execution time an application spends evaluating policies.
- *N* is the number of nodes in the execution tree associated to the application.
- $cost_{policy}$ is the average cost (in time units) of evaluating a policy.
- $cost_{task}$ is the average cost (in time units) of executing a parallel task but without considering the cost of executing its (recursive) subtasks.

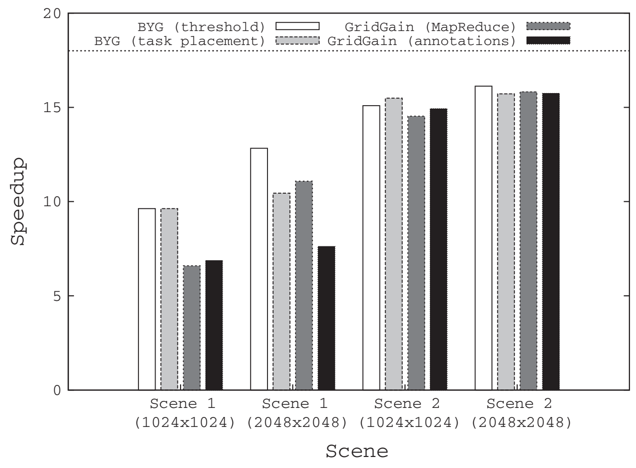
For simplicity, this overhead model makes some assumptions that may not hold in practice, i.e. the formula assumes that the cost of executing task code does not depend on task depth, and all tasks further divide their computation into two subtasks. All in all, as suggested, when using script-based policies the granularity should be controlled. This essentially means keeping *N* below a reasonable limit.

On the other hand, the average overhead of evaluating the Groovy policy with respect to its Python counterpart was 73%, with a standard deviation of 2%. In addition, its curve was steeper. Indeed, Groovy is an agile dynamic language that combines the convenience of scripting with the functionality provided by the Java language itself. Groovy has many of the features of other scripting languages such as Python and Ruby, which are made accessible by relying on a Java-like syntax. However, users should take into account the trade-off between the flexibility offered by Groovy versus its computational requirements. One approach to reduce this overhead would be relying on a lighter version of Groovy so that script parsing and interpretation is less expensive. In principle, this can be solved by incorporating a new scripting engine into the JVM and adding the necessary support to our policy API.

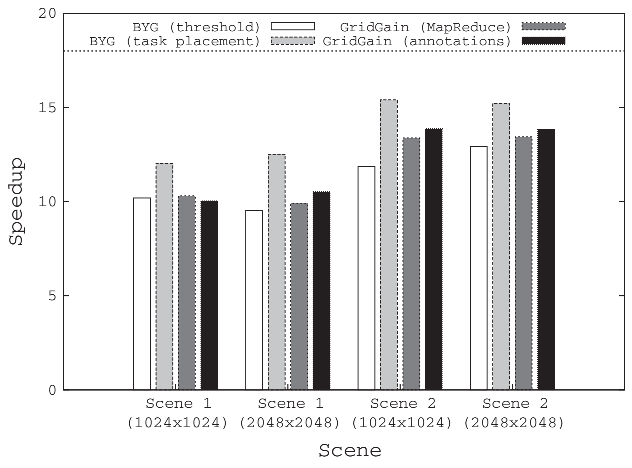
4.2. Real-life applications

To empirically evaluate the applicability of the extensions to BYG described up to now, we conducted several experiments in order to measure the performance that resulted from employing GridGain and BYG for parallelizing two real-world applications, namely ray tracing and sequence alignment. The goal of the experiments was twofold. On one hand, we wanted to determine whether the automatic approach to gridification of BYG via its GridGain connector delivers competitive performance compared to parallelizing applications by hand with GridGain. On the other hand, another goal was to assess the effectiveness of policies for optimizing automatically-gridified applications. Moreover, for an evaluation of the Satin connector, see [28].

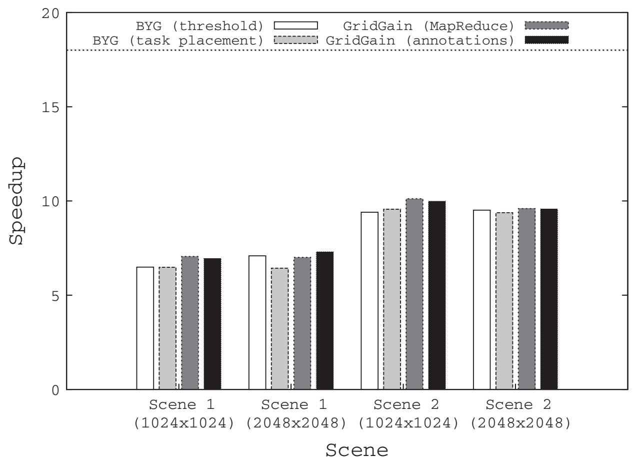
Similarly to the experiments described in [28], we used an emulated Grid setting comprising 15 machines running Mandriva Linux



(a) Fine granularity



(b) Medium granularity



(c) Coarse granularity

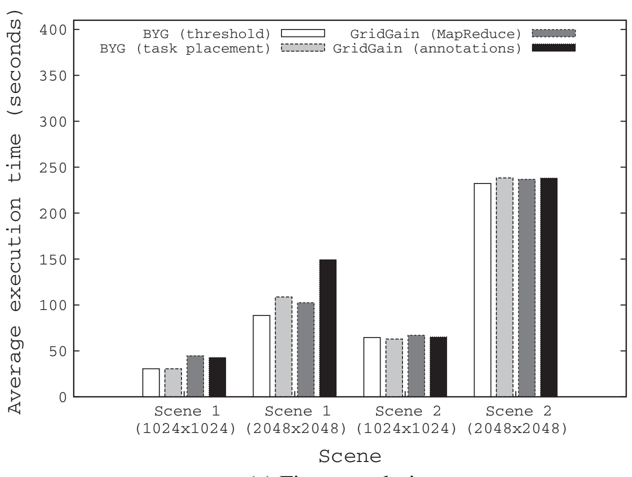
Fig. 7. Ray tracing: Speedup.

2010, Java 5 and GridGain 2.1.0 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. A wide-area Grid on top of this LAN was established by using the WANem version 2.2 [36] WAN emulation software. The resulting Grid was then composed of 3 clusters C_1, C_2 and C_3 by using 4, 5 and 6 of the nodes of the LAN, respectively. Each emulated WAN link had a bandwidth of 1,544 Mbps (i.e. T1 connection) 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. Note that these are network conditions commonly found in Internet-wide Grids. For the sake of fairness, GridGain – and hence BYG – were configured to use the load balancing scheme that best fitted the established Grid, in this case the Round Robin scheduler with the default configuration. All in all, apart from the challenging nature of the execution environment, the test applications had a high cyclomatic complexity, thus they were representative to stress our code analysis mechanisms.

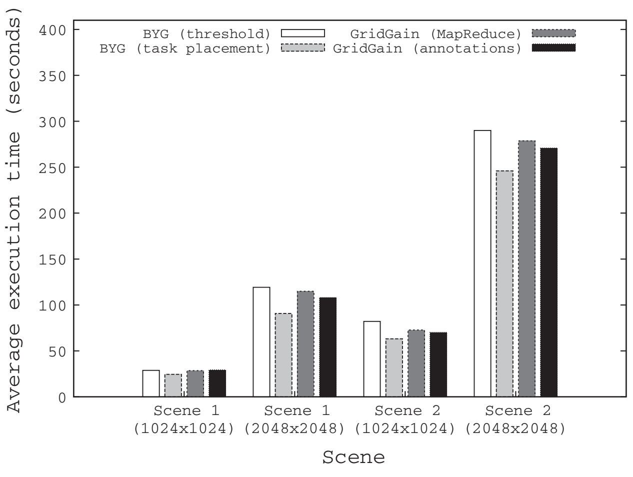
4.2.1. Ray tracing

Ray tracing is a widely-known rendering technique that generates a digital picture from an abstract description of a 3D scene [15]. We based our experiments on an existing D&C parallel ray tracing algorithm from the Satin project, which works by deriving an initial image from the input scene, dividing this image in four portions to recursively apply the algorithm, and then joining the results to build the final picture. From this code, several variants for comparison purposes were coded:

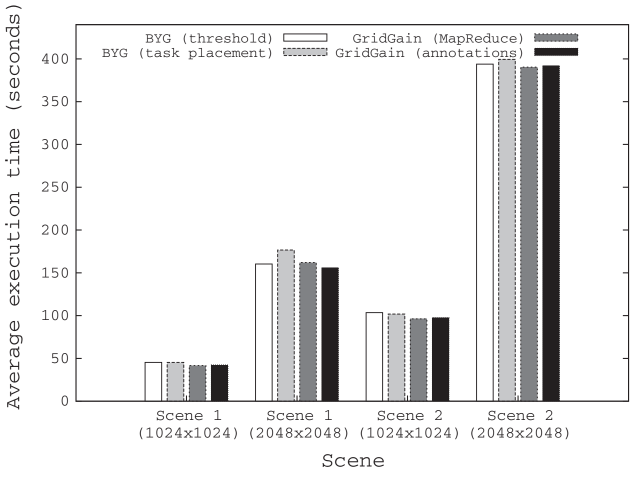
- A variant exploiting the parallel annotations provided by the GridGain platform. With this support, developers annotate their sequential codes for parallelism, and execution of parallelized codes are handled by using a middleware-level mechanism that extends conventional Java futures for distribution.
- A GridGain implementation by altering the original Satin code to exploit the Google's MapReduce parallel programming model [8], which is similar to the master-worker model and is cleanly supported by GridGain. The MapReduce model roughly provides abstractions to generate independent tasks from input data (the map phase) and combines subresults into a larger subresult or result (the reduce phase).
- A BYG variant using a threshold policy to control task granularity. The policy allows the code to spawn tasks provided the size (i.e. width $*$ height) of the subimage being processed is above some given value.
- A BYG variant using a task mapping policy that extends the previous policy with a simple task allocation scheme that places some quadrants belonging to the same subimage in the same cluster. The rationale behind this scheme is to minimize inter-cluster data transfer upon joining subresults.



(a) Fine granularity



(b) Medium granularity



(c) Coarse granularity

Fig. 8. Ray tracing: Average execution time.

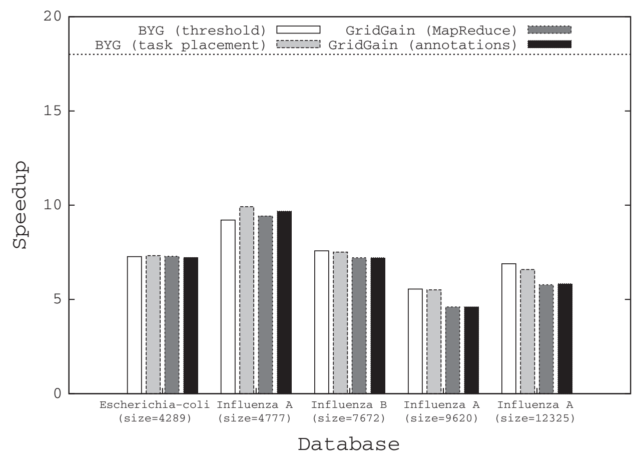
To sum up, we used two hand-coded variants and two automatically-parallelized versions of the application. The GridGain and BYG implementations were obtained by removing from the base Satin code any statement related to parallelism and tuning application execution to derive the sequential D&C counterparts of the application first, and manually or automatically parallelize them later according to the target application structure. As suggested earlier, both BYG variants ended up generating GridGain source code. As the nature of the ray tracing application does not provide an opportunity for using memoization policies, we did not coded the associated variant. Experiments with real benchmark parallel applications that do benefit from memoization policies can be found in [27].

For executing the applications, we used three task granularities: fine, medium and coarse, i.e. about 17, 4 and 1 parallel tasks per Grid node, respectively. As input, we employed two scenes with two different resolutions (1024×1024 and 2048×2048). Unlike the experiments reported in [28], we decided not to process scenes with lower resolutions or spawn more tasks per node as GridGain has proved not to work very well when handling very fine-grained parallelism.

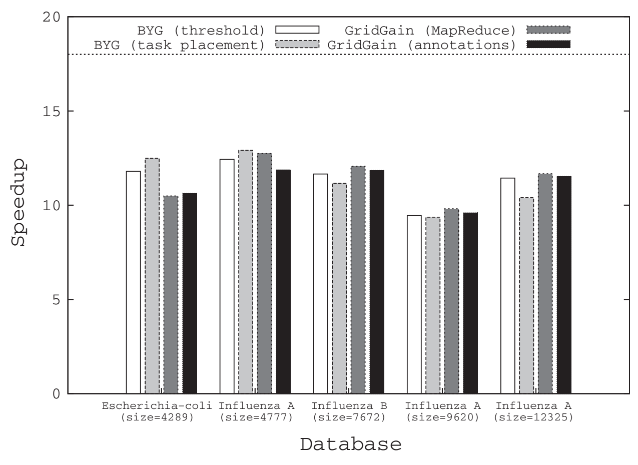
Fig. 8 illustrates the average running time of the applications for 40 executions. The standard deviation was below 11%, which is acceptable considering the random factors that characterize the underlying scheduler and the variability inherent to WAN links in terms of bandwidth and latency. As a complement, Fig. 7 shows the speedup achieved by the different implementations for the various configurations. Speedup was computed by the formula T_{seq}/T_{par}, being T_{seq} and T_{par} the times required to execute the sequential[6] and parallel versions of the ray tracing application, respectively. On the other hand, the theoretical maximum was established not at the number of cluster machines (i.e. 15) but at a value of 18, because the processors of the different machines supported *hyper-threading*, a hardware technology that emulates two processors within the same physical CPU. Then, task schedulers that exploit this feature (typically via threads) such as the one provided by GridGain usually increase CPU performance by 22% at the average. In this sense, the theoretical maximum was approximated as $\rfloor(\#machines + \#machines*0.22)$.

All in all, compared to GridGain, BYG performed rather well, considering that its design goal is not to outperform existing parallel Grid libraries but automating as much as possible parallel programming and therefore the usage of such libraries while achieving competitive performance. Note that the execution times uniformly increased as granularity became coarser for all tests,

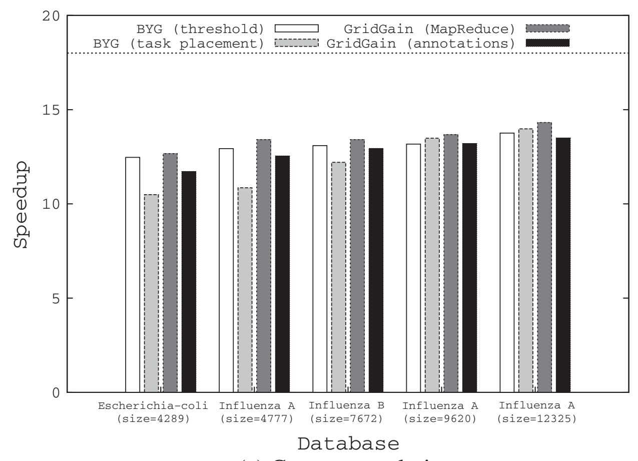
[6] The sequential variant of the application was run on the Grid node featuring the best processing capabilities.



(a) Fine granularity



(b) Medium granularity



(c) Coarse granularity

Fig. 9. Pairwise sequence alignment: Speedup.

which shows a good overall correlation of the different variants. This makes sense because coarser task granularity means that fewer parallel tasks are generated and thus the chance of nodes of being idle or underused during a computation increases.

As depicted in Fig. 8, results show that for the case of fine granularity BYG was able to outperform their two competitors since through its parallelization heuristics in conjunction with either policies BYG achieved performance gains of up to 29% (Scene 1, 1024×1024). For the rest of the inputs, BYG and policies performed very close to the GridGain variants. On the other hand, for medium granularity, the plain threshold policy proved to be insufficient for this test application. However, this did not translate into an irremediable problem, since the task placement policy introduced significant performance improvements. As explained in earlier Sections, switching between policies does not lead to modification of the target parallel application. In this sense, when a specific policy does not deliver the expected results, users can easily supply the same parallel application code with another tuning rule.

Finally, for the coarse granularity, for some scenes the best BYG variants introduced overheads of 1–9% with respect to the most efficient GridGain implementations. As expected, task placement and therefore data locality turned out counterproductive, because the performance benefits of placing a set of related tasks (in this case those that process near regions of the input scene) in the same physical cluster scene became negligible for this experiment because worse load balancing is obtained. Again, the most efficient granularities were fine and medium in the sense they delivered the best data communication over processor usage ratio. This suggests, in principle, that the GridGain connector of BYG is more efficient when coarse task granularities are not used, which in turn means that users should reflect this fact in their policies when adjusting granularity.

4.2.2. Pairwise sequence alignment

Local pairwise sequence alignment is a well-known problem in bioinformatics that involves representing a biological entity (e.g. a gene) in a computer-understandable way – usually strings of characters – and manipulating the resulting representation by using sequence alignment algorithms. These algorithms allow scientists to determine for example whether a newly obtained sample protein sequence represents a virus or not.

As the second real-life test application we then used an existing parallel code from the JPPF project,[7] consisting of a master-worker implementation of an application for aligning protein sequences based upon the Smith-Waterman alignment algorithm [12]. The algorithm roughly outputs a coefficient that represents the level of similarity between two given input sequences by employing a scoring matrix from a set of predefined matrixes. As the original JPPF source code was already parallelized, we first removed the library-

[7] http://www.jppf.org.

dependent parallel code before obtaining its sequential version. Based on this latter, we derived four parallel variants:

- A variant using the parallel annotations provided by GridGain.
- A GridGain implementation based on the Google's MapReduce parallel programming model [8].
- A D&C sequential version, after which we obtained a BYG variant using a threshold policy. Algorithmically, the derived D&C code operated by comparing an input sequence against an entire sequence database by dividing the portions of the data to compare against into two different subproblems until a certain threshold T on the data was reached, which was enforced by the policy.
- Same as before but employing a task placement policy extending the previous policy with an explicit task placement scheme to place computations associated to contiguous data portions of the input database in the same cluster.

Furthermore, we compared five different random sequences against real-world protein sequence databases extracted from the National Center for Biotechnology Information (NCBI) Web site.[8] Upon running the experiments, data was replicated across the nodes of the established Grid. Lastly, sequences in the different databases did not follow any special order and therefore none of the implementations were favored over the others in this respect.

Fig. 10 shows the average running time of the applications for 40 executions. The standard deviation was similar to the case of the experiments presented in the previous subsection. Furthermore, Fig. 9 illustrates the achieved speedups. Unlike ray tracing, in which for each granularity an experiment-wide, or variant-independent number of tasks was employed, for each parallel variant of the sequence alignment application we used a number of subcomputations that depended on input sizes. This avoided to spawn many ultra fine-grained parallel tasks when processing small databases, which would had been unfair to GridGain.

As shown in both Figures, and in opposition to ray tracing, the running times were smaller as the granularity increased. This is interesting as confirms that the algorithmic nature of our second application is quite different compared to the first one, thus ensuring the significance of our experiments. Furthermore, like for the ray tracing application, BYG obtained better performance for the fine granularity, and performed very competitively for the medium granularity. However, again, the GridGain variants were slightly more efficient when using coarse-grained tasks. In general, task placement did not help too much in reducing execution time because, unlike ray tracing, parallel tasks had intuitively a higher degree of independence in terms of processed data. One important lesson learned from this fact concerns the applicability of our tuning support. This is, the results do not imply that task placement policies are not effective but their usage should be decided depending on the nature of parallelized applications, which enforces similar previous findings [27]. In other words, this problem is not exclusive to BYG but also affects in general any explicit parallel programming framework. The policy support discussed so far is not designed to automate application tuning, but to provide a customizable framework that captures common optimization patterns in FJP applications. Then, whether these patterns benefit a particular parallelized application or not depends on its nature. In fact, only a subset of FJP applications are able to effectively take advantage of memoization. The same applies to the rest of the policies.

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

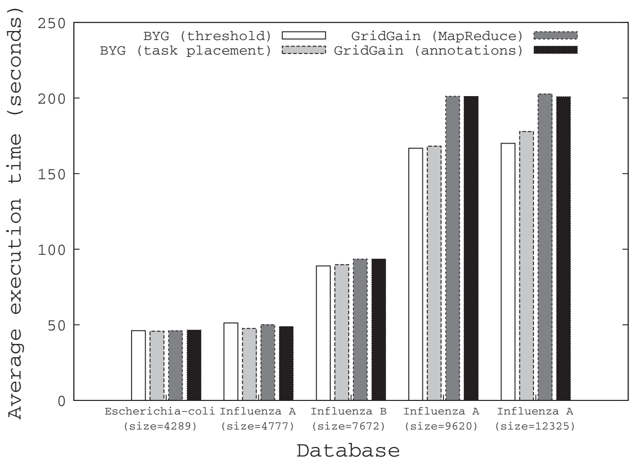
5. Related work

Undoubtedly, MPI and PVM appear themselves as the oldest standards for building engineering and scientific general-purpose parallel applications. When relying on the parallel abstractions proposed by these standards, user applications are parallelized by decomposing them into a number of independent distributed components that communicate between each other via message exchange. As MPI and PVM are standard *specifications*, several implementations for a variety of languages have arisen. In the Java world, MPI is more popular than PVM and is supported by quite a few libraries, being mpiJava [16] and MPJ Express [35] the most recent proposals. Nonetheless, PVM has been also successfully implemented by libraries such as jPVM [39]. Moreover, the JCluster [42] parallel platform, besides providing a distributed programming model based on regular Java threads, provides bindings to both MPI and PVM.

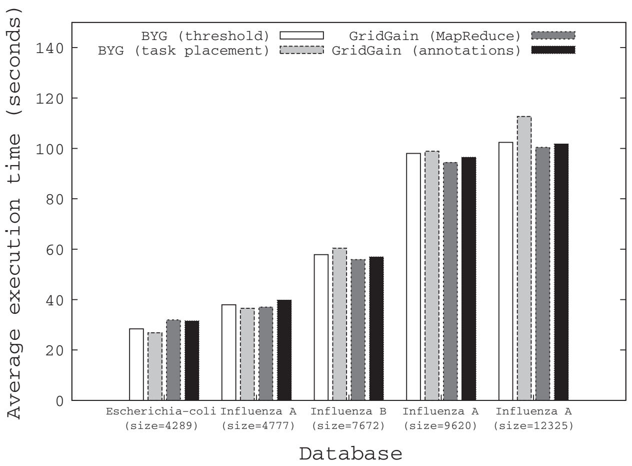
MPI and PVM have on the other hand received much criticism [23] since they are basically low-level parallelization tools that require solid knowledge on both parallel and distributed programming from users. In response, there are Java tools that attempt to address this problem by raising the level of abstraction of the API exposed to users and relieving them as much as possible from performing parallel task creation and coordination. In some cases, these tools also advocate to some forms of semi-automatic parallelism of sequential codes, therefore gradually moving to a new wave of gridification tools that ideally let users to exploit Grids without any programming effort. This is achieved precisely by automating task creation or coordination. Furthermore, other Java-based tools get rid of APIs almost completely and at the same time provide mechanisms that automate some of these aspects, being in this way even closer to this ideal state. Hence, existing tools can be categorized in principle according to two important orthogonal dimensions: the size/complexity of their API, which may be zero (no API is exposed), low, medium and high, and the offered level of automatism for managing parallelism, which may be manual, semi-automatic and (almost) fully-automatic [26]. By manual and semi-automatic we mean that all the effort to turn a sequential application into a Grid-aware source code, without considering configuration and deployment activities, is entirely or partially performed by the user.

ProActive [3] is a rich Java platform for parallel distributed computing that provides *active objects*, or regular Java objects that can migrate between nodes and access computing resources locally. Computations performed by these active objects can be split into several (smaller) subcomputations, which are solved by other active objects. However, managing parallelism still requires manually using its extensive API. Moreover, JavaSymphony [19] is a performance-oriented platform that provides sophisticated middleware-level services for dealing with parallelism and load balancing of Grid applications, and at the same time allows programmers to control such features via API calls placed directly in the application code. Unfortunately, using JavaSymphony, which is an API-inspired parallelization tool, unavoidably requires to learn and manually use their associated APIs within the code of sequential user application to create and synchronize tasks. Similarly, JCluster [42], besides providing Java modules compliant to MPI and PVM, offers an API for parallelizing applications. All in all, the three tools are based on manual parallelism. Besides, both JavaSymphony and JCluster promote threads as the base parallel programming model, which makes application programming, testing and debugging difficult due to the non-deterministic nature of thread execution [23].

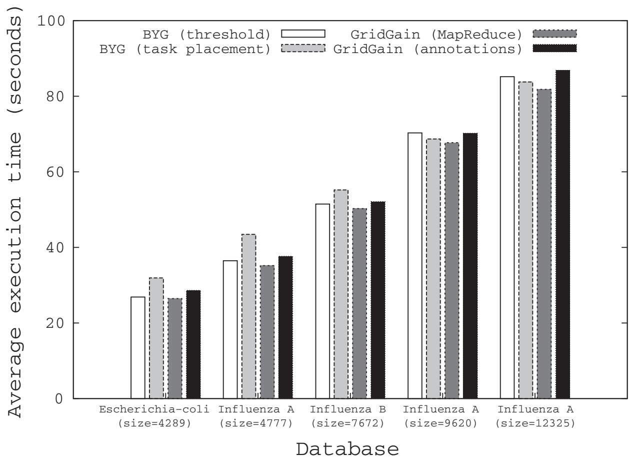
Furthermore, VCluster [43] supports execution of thread-based applications on multicore clusters by relying on a thread migration technique that achieves efficient dynamic load balancing of



(a) Fine granularity



(b) Medium granularity



(c) Coarse granularity

Fig. 10. Pairwise sequence alignment: Average execution time.

threads across the nodes of a cluster. Although based on manual parallelism and a programming model based on threads, the VCluster API is somewhat simpler compared to the tools listed above. Similarly, DG-ADAJ [22] provides a mechanism for transparent execution of multithreaded applications on desktop PC Grids. Interestingly, 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 sequential code. Then, a scheduling heuristic is applied to place mutually exclusive execution paths extracted from the graphs among the nodes of a cluster, thus automatically further exploiting the implicit parallelism of the application. As a consequence of relying on regular threads but automating some aspects of their parallel execution, DG-ADAJ is then based on semi-automatic parallelism.

The two Grid platforms to which BYG provides integration discussed throughout this paper can be also considered as related efforts. The Satin framework [40] avoids the explicit usage of threads while allows parallelizing sequential D&C applications. The user is responsible for implicitly indicating in the application code the points in which forks (i.e. calls to recursive methods) should take place, and explicitly stating joins (i.e. barriers to wait for child computations). Once coded and compiled, Satin further modifies the bytecode of applications to handle the execution of parallel tasks on a Grid. As such, Satin requires partial user intervention in the process of inserting parallel-specific API code prior to fully Grid-enabling their compiled counterpart, which is done by a built-in postprocessor that parallelizes the application bytecode based on specified fork and join points. On the other hand, as discussed earlier, GridGain [13] is a Grid platform providing three parallel development models, i.e. one based on Java futures (shown in Section 2.3), a second using regular Java annotations and a third based on MapReduce (mentioned in Section 4.2). GridGain is aimed among other things at delivering APIs that are easy to learn and use, but these models are based on manual parallelism.

Finally, another line of approaches to gridification aimed at minimizing code modification in the input sequential application and do not rely on parallel API provisioning are those promoting separation of concerns between the functional aspects of the application (pure behavior) and the Grid-specific behavior [14,24]. This is commonly achieved via aspect-oriented programming (AOP) [29] 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. This problem is minimized by several tools such as PAL [7] or the approach described in [11] that are are to a great extent inspired by the parallel programming mechanism promoted by OpenMP [6]. In this sense, these two

Table 4
Analyzed tools: summary.

Tool	API complexity	Approach to parallelism	Base parallel facility
ProActive	high	manual	active object
JavaSymphony	high	manual	thread
JCluster	high	manual	thread
VCluster	medium	manual	thread
DG-ADAJ	zero	semi-automatic	thread
Satin	low	semi-automatic	N/A
GridGain	medium	manual	future, annotation, MapReduce task
Harbulot and Gurd	zero	manual	aspect
Maia et al.	zero	manual	aspect
PAL	low	manual	annotation
Gonçalves and Ferreira Sobral	low	manual	annotation
BYG	*low*	*fully-automatic*	*N/A*

approaches use Java annotations in the source code of sequential applications, thus they provide a more comfortable programming abstraction compared to AOP-based programming tools. Similar to OpenMP and SMP machines, annotated codes are then preprocessed to generate Grid-enabled valid Java code. From a semantic standpoint, these annotations operate just like an API but are far more minimalistic. A key difference of those approaches and OpenMP is, however, that the latter is a *standard, cross-language* set of directives for shared memory parallel programming, whereas the former offer non-standard parallel annotations aimed at parallelism in Java only.

Table 4 summarizes the tools discussed so far. As depicted, BYG differs from the abovementioned efforts since it allows novice developers to automatically introduce parallelism into the compiled version of applications, which avoids the requirement of thinking about how to exploit parallelism in their algorithms, and learning and using parallel programming APIs for managing task synchronization and coordination itself. Parallelism is performed automatically based on heuristics that work by analyzing the input sequential code and generating Grid-aware codes that are in turn prepared to exploit existing Grid APIs. Even when an API is provided by our tool, it is very intuitive and serves as a mean of *optionally* tuning *already* parallelized applications. Not surprisingly, it can be seen from the Table the extent to which conventional language constructs of Java (i.e. threads and annotations) have influenced the development of parallel Java-based platforms. Alternatively, BYG does not explicitly prescribe a specific base parallel facility, and is based on the pervasive and intuitive divide and conquer programming model, an algorithmic abstraction that is present in many real-world problems. Indeed, except for the case of Satin, existing tools are based on parallel programming abstractions that are difficult to manipulate by disciplinary users.

6. Conclusions

In this paper we have described BYG (BYtecode Gridifier), a tool to automatically prepare sequential Java bytecodes to exploit Computational Grids. Particularly, we have focused on describing from a user's standpoint two important extensions of BYG, namely the integration with state-of-the-art Grid schedulers and the incorporation of a policy-based explicit application tuning support. Basically, this support relies on a very simple API that allows users to specify common optimizations for applications without actually modifying their logic. Moreover, optimizations can be coded in Java, Python and Groovy.

BYG offers an alternative balance to the dimensions of applicability, code intrusiveness and expertise that concern parallel programming tools. Good applicability is achieved by targeting Java, FJP and D&C, and leveraging primitives of existing parallel libraries. Low code intrusiveness is ensured by using mechanisms to translate from sequential to parallel code while keeping tuning logic – i.e. statements for optimizing applications – away from this latter. Overall, users such as scientists and engineers can code their algorithms without thinking about parallelism, and then use our tool to Grid-enable their codes and optionally optimize them whenever necessary. Our experimental results confirm that both the heuristics for automatic parallelism and the policy-oriented explicit tuning of BYG are in tandem a viable approach to gridification from a practical perspective.

At present, we are working on tools to make BYG easier to adopt and use. We are developing a prototype implementation of a GUI that lets developers to gridify their applications by graphically selecting target methods and middlewares, and configuring policies. Eventually, this plug-in could also offer proper support for deploying and monitoring the execution of applications by exploiting the analogous services of the selected Grid platform. It is expected that the GUI will also let developers to inspect the execution state of parallelized applications for debugging purposes. In summary, our goal is to supply users with a full-fledged frontend for gridifying D&C applications.

Another line of research involves materializing BYG concepts directly into scripting languages that are commonplace in the scientific and engineering community. The idea is to investigate how to port and exploit the parallelization heuristics of BYG and its associated concepts for such languages beyond script-based policies. As a starting point, we are rethinking our heuristics in the context of the Jython library [20], a Java-based implementation of the Python programming language that includes a module to compile Python source code into Java bytecodes. Interestingly, Jython is compliant to Python 2.5, and supports nearly all of its core standard modules. At present, we are developing a prototype to parallelize applications on multicore servers. Nevertheless, Jython provides extra support that makes it easy to use regular Java classes from within Python scripts. This will be useful to take advantage of the already implemented bindings to existing Grid schedulers shipped with BYG. This work in not being done in isolation but we are also reusing previous advances in the topic reported in the literature [18].

Likewise, we are investigating how to adapt our ideas not only to interpreted languages but also to *compiled* languages that are heavily employed within the scientific and the engineering communities, such as C and C++. Although the algorithms for introducing parallelism and performance tuning are mostly language-independent, their implementation unavoidably requires a language-dependent mechanism for dynamically rewriting compiled sequential codes. In this sense, a technological alternative for implementing this mechanism in C and C++ is to use Dyninst [38], an API that allows on-the-fly modification of native binary codes. Dyninst is a very healthy library (version 7.0 was released on March, 2011), works with a variety of operating systems, and has been already used in similar developments, notably the MATE

(Monitoring, Analysis and Tuning Environment) for parallel applications [5].

Finally, a relatively recent distributed and parallel computing paradigm that is rapidly gaining momentum is Cloud Computing [9,4], which bases on the idea of providing an on-demand computing infrastructure to end users. Typically, users exploit Clouds by requesting from them one or more *machine images*, which are virtual machines running a desired operating system on top of several physical machines (e.g. a datacenter). Interaction with a Cloud is performed via Cloud services, which define the functional capabilities of a Cloud, i.e. machine image management, access to software/data, security, and so forth. Among the benefits of Cloud Computing is precisely a simplified configuration and deployment model compared to clusters and Grids [9], which is extremely desirable for disciplinary users. In addition, Cloud infrastructures intuitively have the capabilities to deliver good performance. Consequently, we will investigate how to adapt BYG to exploit such infrastructures.

Acknowledgments

We thank the anonymous referees for their comments to improve the paper. We thank Cristian Clasadonte for his help managing the computing infrastructure required for the experiments with real-life applications described in this paper. We also acknowledge the financial support provided by ANPCyT through Grant PAE-PICT 2007-02311.

References

[1] Abelson H, Sussman G. Structure and interpretation of computer programs. 2nd ed. Cambridge, MA, USA: MIT Press; 1996.

[2] Alba E, Blum C, Asasi P, Leon C, Gomez JA. Optimization techniques for solving complex problems, parallel and distributed computing. Publishing: Wiley; 2009.

[3] Amedro B, Baude F, Huet F, Mathias E. Combining Grid and cloud resources by use of middleware for SPMD applications. In: 2nd IEEE international conference on cloud computing technology and science (CloudCom), Indianapolis, USA. Los Alamitos, CA, USA: IEEE Computer Society; 2010. p. 177–84.

[4] Buyya R, Yeo C, Venugopal S, Broberg J, Brandic I, computing Cloud, et al. and reality for delivering computing as the 5th utility. Future Gener Comput Syst 2009;25(6):599–616.

[5] Caymes-Scutari P, Morajko A, Margalef T, Luque E. Scalable dynamic monitoring, analysis and tuning environment for parallel applications. J Parallel Distrib Comput 2010;70(4):330–7.

[6] Chandra R, Dagum L, Kohr D, Maydan D, McDonald J, Menon R. Parallel programming in OpenMP. San Francisco, CA, USA: Morgan-Kaufmann Publishers Inc.; 2000.

[7] 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.

[8] Dean J, Ghemawat S. MapReduce: a flexible data processing tool. Commun ACM 2010;53:72–7.

[9] Foster I, Zhao Y, Raicu I, Lu S. Cloud computing and Grid computing 360-degree compared. In: Grid computing environments workshop (GCE '08). Austin, Texas,USA: IEEE Computer Society; 2008. p. 1–10.

[10] Freeh V. A comparison of implicit and explicit parallel programming. J Parallel Distrib Comput 1996;34(1):50–65.

[11] Gonçalves R, Ferreira Sobral J. 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.

[12] Gotoh O. An improved algorithm for matching biological sequences. J Mol Biol 1982;162(3):705–8.

[13] GridGain Systems, GridGain=High Performance Cloud Computing; 2011. <http://www.gridgain.com> [last accessed August 2011].

[14] 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.

[15] Heckbert P, Haines E. A ray tracing bibliography. In: Glassner A, editor. Introduction to ray tracing. Academic Press Inc.; 1989. p. 295–303.

[16] 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.

[17] Herzeel C, Costanza P. Dynamic parallelization of recursive code. Part 1: managing control flow interactions with the continuator. SIGPLAN Notices 2010;45:377–96.

[18] Hinsen K. Parallel scripting with Python. Comput Sci Eng 2007;9:82–9.

[19] Jugravu A, Fahringer T. JavaSymphony, a programming model for the Grid. Future Gener Comput Syst 2005;21(1):239–46.

[20] Juneau J, Baker J, Wierzbicki F, Soto L, Ng V. The definitive guide to Jython: Python for the Java platform. 1st ed. Berkely, CA, USA: Apress; 2010.

[21] Kim J-K, Shivle S, Siegel H, Maciejewski A, Braun T, Schneider M, et al. Dynamically mapping tasks with priorities and multiple deadlines in a heterogeneous environment. J Parallel Distrib Comput 2007;67(2):154–69.

[22] Laskowski E, Tudruja M, Olejnik R, Toursel B. Byte-code scheduling of Java programs with branches for desktop Grid. Future Gener Comput Syst 2007;23(8):977–82.

[23] Lee E. The problem with threads. Computer 2006;39(5):33–42.

[24] Maia P, Mendonca N, 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.

[25] Mateos C, Zunino A, Campo M. JGRIM: an approach for easy gridification of applications. Future Gener Comput Syst 2008;24(2):99–118.

[26] Mateos C, Zunino A, Campo M. A survey on approaches to gridification. Software: Pract Exp 2008;38(5):523–56.

[27] Mateos C, Zunino A, Campo M. An approach for non-intrusively adding malleable fork/join parallelism into ordinary JavaBean compliant applications. Comput Lang Syst Struct 2010;36(3):288–315.

[28] Mateos C, Zunino A, Hirsch M, Fernández M, Campo M. A software tool for semi-automatic gridification of resource-intensive Java bytecodes and its application to ray tracing and sequence alignment. Adv Eng Software 2011;42(4):172–86.

[29] Murphy G, Schwanninger C. Guest editors' introduction: aspect-oriented programming. IEEE Software 2006;23:20–3.

[30] Nakada H. Condor-G Java API; 2008. <http://staff.aist.go.jp/hide-nakada/condor_java_api/index.html> [last accessed May 2011].

[31] Oliphant T. Python for scientific computing. Comput Sci Eng 2007;9(3):10–20.

[32] Papadimitriou S, Terzidis K, Mavroudi S, Likothanassis S. Scientific scripting for the Java platform with jLab. Comput Sci Eng 2009;11:50–60.

[33] Pérez F, Granger B, Hunter J. Python: an ecosystem for scientific computing. Comput Sci Eng 2011;13(2):13–21.

[34] Ropo M, Westerholm J, Dongarra J. Recent advances in parallel virtual machine and message passing interface. In: Proceedings of the 16th European PVM/MPI user's group meeting, Espoo, Finland, September 7–10, 2009. Lecture notes in computer science. Berlin/Heidelberg: Springer-Verlag; 2009.

[35] Shafi A, Carpenter B, Baker M. Nested parallelism for multi-core HPC systems using Java. J Parallel Distrib Comput 2009;69(6):532–45.

[36] TATA Consultancy Services, WANem; 2009. <http://wanem.sourceforge.net> [last accessed April 2011].

[37] Thain D, Tannenbaum T, Livny M. Distributed computing in practice: the Condor experience. Concurr Comput: Pract Exp 2005;17(2–4):323–56.

[38] University of Maryland, Dyninst api; 2011. <http://www.dyninst.org> [last accessed August 2011].

[39] University of Virginia, jPVM; 1999. <http://www.cs.virginia.edu/ajf2j/jpvm.html> [last accessed May 2011].

[40] Van Nieuwpoort R, Wrzesińska G, Jacobs C, Bal H. Satin: a high-level and efficient Grid programming model. ACM Trans Program Lang Syst 2010;32(3):9:1–9:39.

[41] Wang L, Jie W. Towards supporting multiple virtual private computing environments on computational Grids. Adv Eng Software 2009;40(4):239–45.

[42] Zhang B-Y, Yang G-W, Zheng W-M. JCluster: an efficient Java parallel environment on a large-scale heterogeneous cluster. Concurr Comput: Pract Exp 2006;18(12):1541–57.

[43] Zhang H, Lee J, Guha R. VCluster: a thread-based Java middleware for SMP and heterogeneous clusters with thread migration support. Software: Pract Exp 2008;38(10):1049–71.