UDC, DOI: N/A

EasyFJP: Providing Hybrid Parallelism as a Concern for Divide and Conquer Java Applications

Cristian Mateos[1], Alejandro Zunino[1], and Matías Hirsch[2]

[1] ISISTAN Research Institute - CONICET. Also UNICEN University.
Campus Universitario, Tandil (B7001BBO), Buenos Aires, Argentina
cmateos,azunino@conicet.gov.ar
[2] UNICEN University.
matias.hirsch@gmail.com

Abstract. Because of the increasing availability of multi-core machines, clusters, Grids, and combinations of these there is now plenty of computational power, but today's programmers are not fully prepared to exploit parallelism. In particular, Java has helped in handling the heterogeneity of such environments. However, there is a lot of ground to cover regarding facilities to easily and elegantly parallelizing applications. One path to this end seems to be the synthesis of semi-automatic parallelism and Parallelism as a Concern (PaaC). The former allows users to be mostly unaware of parallel exploitation problems and at the same time manually optimize parallelized applications whenever necessary, while the latter allows applications to be separated from parallel-related code. In this paper, we present EasyFJP, an approach that implicitly exploits parallelism in Java applications based on the concept of *fork-join synchronization pattern*, a simple but effective abstraction for creating and coordinating parallel tasks. In addition, EasyFJP lets users to explicitly optimize applications through *policies*, or user-provided rules to dynamically regulate task granularity. Finally, EasyFJP relies on PaaC by means of source code generation techniques to wire applications and parallel-specific code together. Experiments with real-world applications on an emulated Grid and a cluster evidence that EasyFJP delivers competitive performance compared to state-of-the-art Java parallel programming tools.

Keywords: Parallel computing, implicit parallelism, explicit parallelism, Parallelism as a Concern (PaaC), Java, fork-join synchronization patterns, policies.

1. Introduction

The advent of powerful distributed environments such as clusters and Grids equipped with multi-core machines doubtlessly calls for new tools for parallel programming. Consequently, there are libraries and frameworks that allow users to exploit parallelism in their applications. Still, many of these tools remain hard to use for an average programmer, and prioritize performance over other desirable attributes such as code invasiveness and independence of the underlying parallel environment. By code invasiveness we mean mixing application logic with parallel-related statements in programs. Simple parallel programming models are essential to support users not proficient in

parallel concepts, thus helping "sequential" developers to gradually move into the mainstream. Likewise, low code invasiveness and environment neutrality are important given the benefits of hiding parallelism from applications on code maintainability.

In dealing with the software diversity of such environments Java has gained much popularity since it offers platform independence and competitive performance compared to conventional languages [37, 42]. However, most Java parallel libraries have historically focused on running on one specific parallel environment, i.e., either multicore machines or distributed settings. Besides, tools often offer developers APIs and directives for programmatically coordinating parallel subcomputations. This clearly requires knowledge on parallel (and distributed) development, and leads to codes that depend on the library being used, making code maintainability and portability to other libraries an arduous task. Thus, there is not a clear separation between writing application logic and parallelizing it. All in all, parallel programming is nowadays the rule and not the exception. Hence, researchers and software vendors have put down on their agenda the long-expected goal of versatile parallel tools delivering low code invasiveness and development effort.

This paper presents EasyFJP, an approach for parallelizing sequential applications. By "parallelizing" we mean preparing a code to take advantage of a parallel environment. EasyFJP synthesizes semi-automatic parallelism and Parallelism as a Concern (PaaC), through which the difficult and intrusive nature of parallelism is mitigated. Furthermore, EasyFJP exploits the implicit parallelism present in Java-based applications through the concept of *fork-join synchronization pattern*, i.e., a novel parallel abstraction of our own that represent common ways present in existing tools of manually forking and synchronizing subcomputations when parallelizing code. To automate this, EasyFJP provides semi-automatic algorithms and parallel code generation techniques that rely on mechanisms for separation of concerns to isolate application logic from the code in charge of parallelism. Finally, EasyFJP offers an explicit but non-invasive tuning mechanism based on the concept of *policy*, which allows users to specify custom rules to optimize the generated applications at runtime.

EasyFJP builds on previous research carried out by the authors, which was first reported in [27]. In this paper a number of additional contributions are introduced:

- A technology-neutral, conceptualized view of our approach to parallelism as a whole, which in turn could serve to facilitate the materialization of EasyFJP to various programming languages and environments.
- The delineation of the concept of *fork-join synchronization pattern* by presenting the existing patterns and the proposed heuristic algorithms to automate their usage in sequential applications. Unlike application-specific parallel supports such as [38], EasyFJP materializes these patterns for general-purpose recursive codes. This is since FJP is suitable for divide and conquer applications, which is an algorithmic abstraction present in many real-life problems.
- A new type of policy called *task placement policy*, which allows developers to non-invasively control task placement or mapping of unfinished tasks to available executing nodes in Grids. This policy represents a simple mechanism to increase performance by taking into account network characteristics, which is a major source of overheads when executing Grid-aware applications [7].

- Empirical evidence of the feasibility of using our approach to parallelize real-world sequential applications by comparing EasyFJP with other well-known parallel programming models that rely on manual parallelism, such as those based on parallel directives/annotations, and MapReduce [11, 25].
- A rigorous experimental evaluation of the approach by using an emulated computational Grid and a cluster. Indeed, recent literature shows that there is a rising interest in Java-based tools for parallel and distributed computing, but their adoption is delayed due to the lack of up-to-date evaluations of their performance [42].

Particularly, with respect to the last two contributions, the obtained results show that implicit parallelism via fork-join synchronization patterns and policy-oriented explicit tuning, glued together through mechanisms for separation of concerns, is a viable approach to PaaC from a practical perspective. Although our ideas may be applicable to other kind of applications, as suggested earlier we scope our research to divide and conquer applications, since we aim at dealing with massive parallelism.

The rest of the paper is organised as follows. The next Section discusses the most relevant related works by pointing out the novelties of EasyFJP. Then, Section 3 explains the concepts underpinning EasyFJP. After that, Section 4 describes EasyFJP in detail. The Section also includes source code examples to illustrate our approach. It is worth noting that, since the advances presented in this paper not only generalize but also complement our previous efforts, we have deliberately included some of the explanations already reported in [27] to make this article self-contained and therefore more readable. Section 5 reports an evaluation of EasyFJP with two-real world applications, namely sequence alignment and ray tracing, on an emulated computational Grid and a cluster. Finally, Section 6 concludes the paper.

2. Parallelism in Java: background

In light of the increasing amount of available hardware, many Java tools for implementing CPU-hungry parallel applications have been proposed.

2.1. Multi-core programming

Doug Lea's framework [23] is a Java API that offers functionality for queuing and synchronizing concurrent subcomputations. Alternatively, JCilk [10] supplies Java with the *spawn* and *sync* library primitives. Each parallel method is associated with two clones, one used in the common case where serial semantics suffice, and another executed when parallel semantics are required. JCilk obeys the ordinary semantics of the try/-catch construct when executing on a single core CPU, but causes subcomputations to abort when an exception occurs on a multi-core machine. Furthermore, JAC [17] aims at separating application logic from thread declaration and synchronization via Java annotations, emphasizing on removing the differences between sequential and concurrent codes. Duarte et al. [13] address the same goal by automatically deriving thread-enabled codes from sequential ones based on algebraic laws. Similarly, JOMP [5] is compliant to OpenMP [9], a popular set of standard method-level/sentence-level directives and library routines for shared memory parallel programming.

2.2. Cluster and Grid programming

JR [8] provides a rich concurrency model supporting remote JVM and object creation, asynchronous communication and rendezvous. JR codes are translated into regular Java codes. JCluster [46] supports the execution of task-oriented parallel applications in heterogeneous clusters. Tasks are scheduled according to the novel transitive random stealing algorithm. Moreover, Satin [44] is a library for parallelizing divide and conquer codes on LANs and WANs that follows the semantics of JCilk. A programmer marks through API classes and interfaces the application methods that must run in parallel. Satin then Grid-enables the application by modifying its compiled code. JavaSymphony [21] is another platform that features a semi-automatic execution model that transparently deals with migration, parallelism and load balancing of Grid applications, and allows programmers to control such features via API calls embedded in their codes.

Furthermore, VCluster [47] executes thread-based applications on clusters. Threads migrate between nodes for load balancing purposes. Inter-thread communication is performed through *virtual channels*, which isolate threads physical location (i.e., machine). ProActive [4] allows developers to program parallel applications composed of *active objects*, which have migratory capabilities. Active objects asynchronously or synchronously communicate with other active or regular objects via method calls. Active object creation, communication and mobility are programmatically performed via an API. In addition, JGRIM [26, 28] gridifies applications by non-invasively attaching Grid concerns such as resource brokering, mobility and parallelism through Dependency Injection [20], which allows component-oriented Java applications [35] to be seamlessly supplied with middleware-level components that implement those concerns. Last but not least, with respect to the plethora of tools for building classical master-worker applications, two representative examples are GridGain [16] and JPPF [40].

2.3. Summary of the discussed tools

Parallel programming is commonly classified into two categories: automatic (or implicit) and manual (or explicit) [45, 15]. The former allows developers to write parallel applications without any concern about parallelism, which is performed by the underlying runtime system in an automatic way. However, by following this approach performance may be suboptimal. Alternatively, explicit parallelism supplies APIs or directives for initiating and coordinating subcomputations. Developers have thus more control over parallel execution to implement efficient applications, but the burden of managing parallelism falls on them. From now on, “automatic” and “implicit”, and “manual” and “explicit”, will be used interchangeably throughout the rest of the paper.

Many of the above efforts are inspired by explicit parallelism. Despite its performance, a negative side-effect of using traditional explicit parallelism is that parallelizing applications requires learning the concepts and the features of the parallel tool being used, which may not be easy for an average programmer. From a software engineering standpoint, parallelized codes are hard to maintain and port to other libraries. Lastly, using these approaches lead to parallel codes that contain not only statements for managing subcomputations but also for tuning the application. This makes such tuning logic

obsolete when the application is ported to a different environment, e.g., from a cluster to a Grid. This decision could be taken for example for scalability purposes.

An alternative approach to traditional explicit parallelism is to treat parallelism as a concern, thus avoiding mixing application logic with code implementing parallel behavior. This idea is followed by several Java tools which partly rely on mechanisms for separation of concerns, e.g., class and method-level code annotations (JAC, GridGain), metaobjects (ProActive), and Dependency Injection (JGRIM). As one may expect, many other efforts support the same idea through AOP, which is the most widely known technique in this line, and *skeletons*, which capture recurring parallel application structures such as pipe and master-worker in an application-agnostic way. These structures, which are analogous to object-oriented design patterns, are instantiated by composing wrapped sequential codes or specializing framework classes [2, 39].

In our view, current approaches pursuing PaaC fall short with respect to applicability, code intrusiveness and expertise. Tools designed to exploit single machines are in general not applicable to distributed settings, whereas approaches designed to exploit such settings experience overheads when used in multi-core machines due to their distributed nature. Moreover, approaches based on code annotations require explicit modifications to insert parallelism and application-specific optimizations that obscure the final code. Metaobjects and specially AOP have proven to cope with this problem, but at the expense of demanding programmers to learn another programming paradigm. Lastly, tools providing support for various parallel patterns feature good applicability in respect to the variety of anatomies of parallel applications that are supported. However, such approaches require knowledge on parallel notions (e.g., parallel patterns) and altering application logic after parallelization demands first to understand the design of the produced parallel code.

We argue that PaaC should be further exploited to offer novice users a hybrid approach to parallel development that takes the simplicity of implicit parallelism, and the flexibility and efficiency of explicit parallelism. In this sense, we propose EasyFJP, an approach to parallelism for Java that *implicitly* leverages parallelism in sequential applications, and at the same time allows users to *explicitly* tune the parallelized codes without affecting the application logic. By drawing a parallel with existing proposals, EasyFJP borrows ideas from the implicit approach to parallelism followed by functional languages designed for multi-core CPUs –e.g., Erlang [3], Haskell [19]– that exploit the inherent concurrency present in applications, and the explicit approach taken by existing Java-based parallel and distributed programming tools promoting separation of concerns, but by proposing a simpler approach to PaaC for a family of applications, i.e., divide and conquer codes. Furthermore, execution of parallel applications is performed by leveraging the schedulers of existing Java parallel libraries.

EasyFJP offers users who are not experienced in parallel programming means for parallelizing applications by adopting a programming model that provides opportunities for implicit nevertheless versatile forms of parallelism, and using a generative programming approach to build parallel codes that reuse existing parallel/distributed programming libraries. Developers can then optimize generated parallel and distributed codes. The next section presents the EasyFJP approach in detail.

3. Fork-join parallelism

Fork-join parallelism (FJP) is a simple but effective technique through which parallelism is expressed via two primitives: *fork*, which starts the execution of a code fragment (commonly a procedure or a method) in parallel, and *join*, which blocks its caller until the execution of the code fragment finishes. FJP represents a high level mechanism to handle threads in programs, whose direct usage has received criticism due to the inherent complexity of developing, testing and debugging threads [24]. In fact, Java, which has offered threads as first-class citizens for years, includes now an FJP framework for multi-core CPUs[3]. Indeed, easy-to-use programming models like FJP are of major importance as they can boost the performance of today's sequential applications without the pressing need for a solid background on parallel programming.

Broadly, FJP is useful for execution environments where the notions of "task" and "processor" exist. For instance, forked tasks can be run among the nodes of a cluster, thus improving performance and scalability. Some years ago, Computational Grids [14], which arrange resources from dispersed sites, have emerged as another exciting environment for parallel computing. Interestingly, multi-core CPUs, clusters and Grids alike can execute FJP tasks, because 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 suggests that the same FJP application may be run in either environments, provided there is a scheduler aware of the specifics of the underlying support. Then, a requirement for higher performance on a multi-core FJP application may be fulfilled by gridifying it.

Broadly, current Java parallel libraries relying on task-oriented execution models offer primitives to fork one or many tasks at once. These tasks are explicitly mapped through API calls to library-level execution units. For example, the JPPF framework provides a *job* abstraction, implemented by the JPPFJob class, which serves as a container of one or more parallel tasks. There are however operational differences among libraries concerning the primitives to synchronize subcomputations. From Section 2 we conclude that these primitives follow one of two *fork-join synchronization patterns*: single-fork join (SFJ) and multi-fork join (MFJ). The former represents one-to-one relationships between fork and join points, i.e., a programmer must block its application to wait for the result of each task. Alternatively, MFJ models many-to-one relationships, thus the programmer waits for the results of the tasks launched up to a synchronization call. For example, in the following code snippet, two SFJ calls are necessary to safely access the results of $task_1$ and $task_2$:

```
public class SomeClass{
 public void someMethod(){
  ...
  API.fork(task1);
  API.fork(task2);
  ...
  API.sfj(task1); /* Block until task1 finishes */
  ...             // Access the result of task1
```

[3] http://openjdk.java.net/projects/jdk7/features

```
  API.sfj(task2); /* Block until task2 finishes */
  ...             // Access the result of task2
 }
}
```

, whereas the next code achieves the same behavior with one MFJ call:

```
public class SomeClass{
 public void someMethod(){
  ...
  API.fork(task1);
  API.fork(task2);
  ...
  API.mfj([task1, task2]); /* Block until task1, task2 finish */
  ...                      // Access either results
 }
}
```

Examples of Java-based parallel libraries that support such synchronization patterns are GridGain (SFJ), JPPF (SFJ), ProActive (SFJ and MFJ) and Satin (MFJ), which developers take advantage of through certain API calls. Sadly, this requires to learn a parallel API, and ties the code to the library at hand. Moreover, managing synchronism for real-world applications with complex algorithmic structures is time-consuming, and even more important, error prone, as debugging concurrent programs has been historically conceived as a notoriously hard task [29].

4. Semi-automatic FJP as a concern: the EasyFJP project

Intuitively, FJP is suitable for parallelizing divide and conquer (D&C) applications. D&C is a natural way of solving a problem by recursively breaking it down into several subproblems until trivial subproblems are obtained. Small, non-dividable subproblems are commonly solved by calling a fragment of sequential code. The solutions to the different subproblems are then combined to solve the whole problem.

Our ongoing EasyFJP project proposes source code analysis algorithms and generation techniques to automate the task of introducing SFJ and MFJ into sequential D&C codes. Basically, these algorithms exploit the implicit fork-join structure present in a sequential D&C application, whereas the techniques generate an FJP version to leverage a parallel library of the user's choice. Parallel code generation considers the synchronization support of the target library, i.e., SFJ or MFJ. Central to EasyFJP is a semi-automatic parallelization process that outputs library-dependent parallel applications with hooks for attaching user-provided optimizations called *policies*. This process is illustrated in Figure 1. In this context, by parallel application we mean an application comprising parallel entities that do not share memory and require no synchronization between each other during their execution other than joins.

In the first step (Section 4.1), the source code of the application is analyzed to spot the points within the target method that perform recursive calls and accesses to their results. These calls and accesses together form the task-result dependencies of the

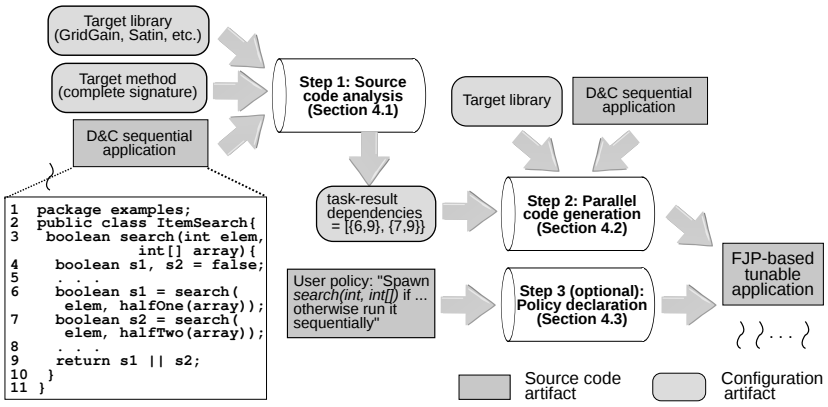


Fig. 1: EasyFJP: Parallelization process

method. Based on these dependencies, the points in which library-dependent synchronization code is to be inserted in the method are identified and passed on to the next step. In this way, the spotted dependencies are ensured upon parallelization to keep the correctness of the resulting application. Before processing a D&C sequential application with EasyFJP, programmers have to follow a simple code convention on their sequential codes, namely assigning the results of recursive calls to local variables.

Targeting parallel libraries featuring SFJ simplifies the task of automatically identifying synchronization points and hence inserting library-specific code to handle dependencies, since the accesses to task results can be (usually) directly replaced by a proper blocking API call. This is harder with MFJ, as it is necessary a deeper code analysis to consider the structure of sentences and variable scopes, while ensuring the spotted dependencies and minimizing the inserted parallel blocking calls. Either synchronization algorithms behave heuristically, emulating a clever human developer while keeping the correctness of the produced code.

The second step (Section 4.2) involves the generation of the parallel code itself through *builders*, which are components that take advantage of the primitives of the target parallel library. Builders also insert glue code to dynamically evaluate the optimizations potentially defined at step 3. Lastly, builders adapt the code to the application structure prescribed by the tarjet library. This includes extending from certain API classes and generating extra code artifacts, among others.

A challenging issue in this regard concerns adapting code to the parallel programming model of the target libraries. For example, targeting D&C libraries 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. However, targeting 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 builders must somehow "flat" the task structure of the input application. An example of such a library is GridGain. We have nevertheless

developed builders for GridGain, and Satin, as will be explained in further paragraphs. Furthermore, we are studying other libraries to detect more adaptation scenarios.

Lastly, at step 3 (Section 4.3), programmers can optionally customize their parallel applications for efficiency purposes via a non-intrusive tuning mechanism based on *policies*. Conceptually, a policy is a rule that controls the amount of parallelism of an application. It is implemented as a user-supplied class that specifies whether to fork a recursive call or run it sequentially instead. For instance, ItemSearch (Figure 1) could be made forking the recursive calls to search provided the length of the input unsorted array is above some threshold.

Policies are associated to fork points through external configuration, and therefore they can be modified and switched without altering the application code. Policy usage is not mandatory for parallelizing applications. In addition, the separation promoted by this mechanism –which is inspired by the concept of separation of concerns– between the tasks of writing application logic and tuning it contributes to the application development process, as these two activities can be carried out independently by programmers with different skills. Moreover, our tuning framework allows developers to specify policies based on the nature of their applications and the execution environment, e.g., using memoization or avoiding many forks with large-valued parameters in a high-latency network.

One may argue that an alternative rule for paralleling D&C code is to directly rely on the built-in FJP framework of Java 7. As such, this framework (and hence the parallelized code) is portable to any implementation of Java. However, the framework does not support distributed computing. Therefore, to support distributed task execution, a parallel library that conforms to this framework API is needed. Even when it is not clear whether adapting parallel libraries to support Java 7's FJP is more difficult than to produce an EasyFJP builder for these libraries or not, not all library providers are interested in being compliant to this framework (e.g., JPPF and Satin). EasyFJP then bridges the gap between D&C sequential codes and parallel libraries relying on fork-join synchronization but not necessarily compliant to Java 7's FJP.

4.1. Step 1: Source code analysis

Before feeding EasyFJP with a sequential application, the result of each recursive method must be stored in a local variable, which allows EasyFJP to automatically spot task--result dependencies and determine the points in which synchronization barriers are needed. This, in turn, ensures that recursive results are always available before they are accessed[4]. In case a programmer targets a library supporting SFJ such as Doug Lea's framework or GridGain, the resulting join points are in fact the points in which those local variables are read. When generating code for a parallel library based on MFJ such as Satin, a smarter source code analysis is necessary to minimize the number of inserted synchronization barriers and hence parallel code. The algorithms for spotting join points based on SFJ and MFJ are shown in Algorithm 1 and Algorithm 2, respectively.

[4] The source code conventions required by the EasyFJP approach are feasible to be automated through proper IDE support [27].

Identifying SFJ-based synchronization points. The algorithm for spotting SFJ-based synchronization points works by walking through the instructions of a method and detecting the points in which a local variable is either *defined* or *used*. A local variable is defined and thus becomes a *parallel variable* when the result of a recursive method is assigned to it. Conversely, a parallel variable is used when its value is read. When executing in parallel, to work properly recursive methods can read parallel variables provided a join has been previously issued. Based on the identified join points at this step, EasyFJP modifies the source code so as to ensure that a library-specific join primitive is called between the definition and use of any parallel variable, for any execution path between these two points. Any regular local variable that does not represent results from parallel computations, or non-parallel, is naturally ignored by the algorithm.

The algorithm operates on the nodes of the AST tree derived from the source code of the input method. As such, this tree represents the different scopes of the method, i.e., the root scope given by the method itself and the scopes resulting from container sentences such as loops, conditionals, etc. Furthermore, the arcs of the three represent the relationships between the scopes. Algorithm 1 shows the process of identifying both the fork and the join points of a D&C method. Fork points are obtained by traversing the sentences of the tree that is derived from the method in a depth-first fashion and looking for definitions of parallel variables. The output of this analysis feeds another process in charge of spotting the points of the method in which synchronization barriers are to be inserted. Table 1 lists the helper functions of this algorithm.

Table 1: Algorithm for spotting SFJ-based synchronization points: Helper functions

Signature	**Functionality**
getParallelVar (aSentence,rootScope)	Checks whether *aSentence* is an assignment of a recursive call to a parallel variable. If so, the name of the parallel variable defined is returned, otherwise an empty result is returned.
getParallelVar (aSentence)	Returns the name of the parallel variable defined in *aSentence*.
getFirstUse (varName,aSentence)	Returns the first subsequent sentence of *aSentence* that uses *varName*. If no such a sentence if found, an empty result is returned.
getScope (aSentence)	Returns the scope to which *aSentence* belongs. Sentences belong to one scope only; if a parent scope S_P has a child scope S_c, a sentence of S_c does not belong to S_P.
checkIncluded (aScope,anotherScope)	Checks whether *aScope* is the same scope as *anotherScope* or is a descendant of it by inspecting the corresponding tree.

Identifying MFJ-based synchronization points. Algorithm 2 summarizes the process of identifying the MFJ-based points (*joinPoints*) of a D&C method. To detect fork points, the first procedure included in Algorithm 1 is used. Like the previous algorithm,

Algorithm 1 Spotting SFJ-based points

```
procedure IDENTIFY FORK POINTS (rootScope)
   forkPoints ← empty
   for all sentence ∈ TRAVERSE DEPTH FIRST (rootScope) do
      varName ← GET PARALLEL VAR(sentence,rootScope)
      if varName ≠ empty then
         ADD ELEMENT (forkPoints,sentence)
      end if
   end for
   return forkPoints
end procedure
procedure IDENTIFY JOIN POINTS (rootScope, forkPoints)
   joinPoints ← empty
   for all sentence ∈ forkPoints do
      varName ← GET PARALLEL VAR(sentence)
      currSentence ← sentence
      scope ← true
      repeat
         useSentence ← GET FIRST USE(varName,currSentence)
         if useSentence ≠ empty then
            useSentenceScope ← GET SCOPE (useSentence)
            varNameScope ← GET SCOPE (sentence)
            if CHECK INCLUDED (useSentenceScope,varNameScope) then
               ADD ELEMENT (joinPoints,useSentence)
               currSentence ← useSentence
            end if
         else
            scope ← false
         end if
      until scope ≠ true
   end for
   return joinPoints
end procedure
```

the MFJ-based algorithm operates on a tree-based representation of the source code of the input method as well. However, it is less intuitive, since it is in fact an heuristic that aims at inserting a minimal number of synchronization barriers, as we detail below. In other words, the heuristic pays attention to both correctness and efficiency aspects.

The algorithm maintains a map with the parallel variables and their associated state (SAFE or UNSAFE) per scope. The former means that up to the current analyzed instruction a parallel variable is safe to use and a synchronization barrier is not needed. In opposition, the latter means that a barrier from where the variable is defined is needed. The algorithm takes into account the scope at which parallel variables are defined and used, i.e., it computes the state of each variable according to the state it has within the (scope) node of the tree where the variable is read and the state of the same variable within the ancestors of that node. Table 2 lists the helper functions of the algorithm.

Algorithm 2 Spotting MFJ-based points

```
procedure IDENTIFY JOINPOINTS (rootScope)
   joinPoints ← empty
   for all sentence ∈ TRAVERSE DEPTHFIRST (rootScope) do
      varName ← GETPARALLELVAR(sentence, rootScope)
      if varName ≠ empty then
         currentScope ← GETSCOPE (sentence)
         if BEINGUSED (varName, sentence) = true then
            if GETFIRSTSTATE (varName, currentScope) = UNSAFE then
               SYNCVARSINSCOPE (currentScope)
               ADDELEMENT (joinPoints, sentence)
            end if
         else if BEINGDEFINED (varName, sentence) = true then
            DESYNCVARUPTOROOT (varName, currentScope)
         end if
      end if
   end for
   return joinPoints
end procedure
```

Let us apply the algorithm to the sequential method shown below. The method contains one non-parallel variable (nonParallelVar) and two parallel variables (varA and varB). The points in which a call to an MFJ-based barrier are needed are explicitly indicated in the code. Table 3 shows the state of varA and varB within their associated scopes as the analysis progresses.

```
1  public String D&CMethod() { // Scope 1
2   ...
3   boolean nonParallelVar = (Math.random() > 0.5) ? true:false;
4   String varA = D&CMethod();
5   if (!nonParallelVar) { // Scope 1.1
6    String varB = D&CMethod();
7    if (Math.random() > 0.5) { // Block 1.1.1
8     // An MFJ should be inserted here
9     System.out.println(varB);
10    varA = D&CMethod();
11   }
12  }
13  if (nonParallelVar) { // Scope 1.2
14   // An MFJ should be inserted here
15   System.out.println(varA);
16  }
17  ...
18 }
```

The algorithm iterates the instructions up to line 4, in which varA is defined. Hence, varA becomes UNSAFE in scope 1 (see stage #1 in Table 3). At line 6, varB is defined within scope 1.1, which makes it UNSAFE in scope 1.1 and its parent scope 1 (stage #2). At

Table 2: Algorithm for spotting MFJ-based synchronization points: Helper functions

Signature	**Functionality**
getParallelVar (aSentence)	Checks whether *aSentence* references –i.e., either defines or uses– a parallel variable. In such a case, the variable name within the method is returned. Otherwise, an empty result is returned.
getScope (aSentence)	Returns the scope to which *aSentence* belongs.
beingUsed (varName,aSentence)	Checks whether the *varName* parallel variable is being read. Analogously, beingDefined checks whether a parallel variable is assigned the result of a recursive call. For container sentences, both functions check whether the variable is accessed in the header of the sentence, but not in the body.
getFirstState (varName,scope)	Traverses the scope tree starting from the node represented by *scope* upwards looking for the occurrence of a parallel variable *varName* in any of the variable maps of these scopes. When the variable is first found, the function returns the state it has in the variable map of the scope it was first encountered.
syncVarsInScope (scope)	Sets to SAFE the state of all parallel variables contained in *scope* (encountered up to the current analyzed sentence) as well as the ancestors of *scope*. The resulting pairs [varName,SAFE] are only put into the map of *scope*.
desyncVarUpToRoot (varName,scope)	Sets the state of a specific parallel variable to UNSAFE from a given scope up to the root scope. This means that the variable becomes UNSAFE in *scope* as well as all its ancestor scopes.

line 9, varB is used within scope 1.1.1. Its first occurrence is encountered in the parent of scope 1.1.1 as UNSAFE. All parallel variables in the variable maps of scope 1.1.1 (none) and its ancestors (varA and varB) are set to SAFE in scope 1.1.1, and the line right before line 9 is regarded as a join point (stage #3). At line 10, another definition of varA is found, which makes the variable UNSAFE in scopes 1.1.1, 1.1 and 1 (stage #4). At line 15, varA is being used within scope 1.2. According to its parent scope 1, the first state of this variable is UNSAFE. This causes to set to SAFE in scope 1.2 all variables found in the maps of scope 1.2 (none) and its ancestors (varA and varB), and to regard the line right before line 15 as a join point (stage #5).

4.2. Step 2: Parallel code generation

Based on the synchronization information obtained at step 1, the source code and configuration of the input application, and the parallel library selected by the developer, EasyFJP generates parallel source code. For each class of the input application, EasyFJP creates a *peer* class that exploits the target parallel API. Then, sequential classes and peers are seamlessly "wired" at load time by employing a simple bytecode rewriting

Table 3: Variable maps and MFJ-based synchronization points of D&CMethod

Stage	Scope	Variable map	joinPoints
1	1	{[varA,UNSAFE]}	-
	1.1	-	
	1.1.1	-	
	1.2	-	
2	1	{[varA,UNSAFE], [varB,UNSAFE]}	-
	1.1	{[varB,UNSAFE]}	
	1.1.1	-	
	1.2	-	
3	1	{[varA,UNSAFE], [varB,UNSAFE]}	line 9
	1.1	{[varB,UNSAFE]}	
	1.1.1	{[varA,SAFE], [varB,SAFE]}	
	1.2	-	
4	1	{[varA,UNSAFE], [varB,UNSAFE]}	line 9
	1.1	{[varA,UNSAFE], [varB,UNSAFE]}	
	1.1.1	{[varA,UNSAFE], [varB,SAFE]}	
	1.2	-	
5	1	{[varA,UNSAFE], [varB,UNSAFE]}	lines 9, 15
	1.1	{[varA,UNSAFE], [varB,UNSAFE]}	
	1.1.1	{[varA,UNSAFE], [varB,SAFE]}	
	1.2	{[varA,SAFE], [varB,SAFE]}	

technique. This essentially aims at avoiding modifying the source code of the original classes while supporting parallelism for them through those peers.

This technique uses the java.lang.instrument package, a built-in API of Java that defines hooks for modifying classes at load time. The package is intended to be extended through special user libraries –regular JAR files (Java ARchive)– called Java agents. These agents customize the class loading process and are accessed by the JVM every time an application requests to load a class. In EasyFJP, the classes that are subject to modification are the ones configured by the user as targets for parallelization.

When rewriting a sequential class for such purposes, EasyFJP replaces the body of its D&C method with a stub that delegates its execution to its parallel counterpart in the associated peer. For example, let us consider the recursive version for computing the n_{th} Fibonacci number, whose code is as follows:

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

```
}
```

Then, the fibonacci method of the FibApp class is dynamically rewritten as follows, so that its computation is performed by the peer (FibApp_Peer):

```
public class FibApp{
 public long fibonacci(long n){
  FibApp_Peer peer = new FibApp_Peer();
  copyProperties(this, peer);
  // Assuming we are relying on SFJ through GridGain
  ExecutorManager m = ManagerFactory.getExecutor("GridGain");
  return (long)m.execute(peer, "fibonacci", new Object[]{n});
 }
}
```

The instance variables of the peer, which may be used by the computation, are instantiated via Java reflection from the running sequential object. Basically, copying properties is a generic procedure that is possible thanks to the uniformity provided by following the getters/setters convention of the well-known JavaBean specification [41], to which sequential classes must also be compliant to. Finally, ExecutorManager represents the EasyFJP API class that communicates with the library-level support that executes peers by performing the corresponding parallel library-specific initialization and disposal activities.

Let us illustrate the peers generated by EasyFJP based on the GridGain library, which offers SFJ-based synchronization and is currently supported by EasyFJP:

```
1  import org.gridgain.grid.Grid;
2  import org.gridgain.grid.GridFactory;
3  import org.gridgain.grid.kernal.executor.*;
4  import org.gridgain.grid.GridTaskFuture;
5  import java.util.concurrent.Callable;
6  // Peer
7  public class FibApp_Peer implements java.io.Serializable{
8   // Properties are copied as is from the original class
9   ...
10  public long fibonacci(long n){
11   return fibonacci(n, initContext(...));
12  }
13  // The GridGain-enabled method
14  public long fibonacci(long n, ExecutionContext ctx){
15   ...
16   Grid grid = GridFactory.getGrid();
17   GridExecutorCallableTask ex = new GridExecutorCallableTask();
18   GridTaskFuture<long> f1future = grid.execute(
19    ex, new FibAppTask(this, updateContext(ctx, ...), n - 1);
20   GridTaskFuture<long> f2future = grid.execute(
21    ex, new FibAppTask(this, updateContext(ctx, ...), n - 2);
22   return f1future.get() + f2future.get();
23  }
24 }
25 // Subcomputation
```

```
26 public class FibAppTask implements Callable{
27  // Instance variable declarations
28  ...
29  FibAppTask(FibApp_Peer peer, ExecutionContext ctx, long p0){
30   // Copy arguments into instance variables
31  }
32  // Setters/getters for "peer", "ctx" and arguments
33  ...
34  public Serializable call(){
35   return getPeer().fibonacci(p0, ctx);
36  }
37 }
```

As shown in the code, the peer contains a proxy method (lines 10-12) that invokes the actual parallelized method (lines 14-23), whose code has been derived from the original fibonacci method but modified to include GridGain forks and joins (lines 18-21 and 22, respectively). Particularly, GridGain materializes the SFJ pattern by extending the conventional *future* construct from the Java concurrency API to support task asynchronysm and distribution. In a broad sense, a future is an abstraction that allows programmers to represent and manipulate an individual asynchronous computation. For the sake of simplicity, the code shown does not exploit the latest version of the GridGain API (i.e., 4.1) since it is fairly more verbose than previous versions.

Instances of FibAppTask carry out the subcomputations by calling fibonacci(long, ExecutionContext) on individual branches of the execution tree. Besides, the peer keeps track of the depth of the tree at runtime. This information, together with the method parameters are encapsulated in an ExecutionContext object, which is used to feed policies by further modifying the source code of the newly generated parallel method. The additional modifications are essentially glue code for invoking policies by passing along context information. It is out of the scope of this paper to detail such code modifications, which can be found in [27].

Besides automating the use of GridGain and its SFJ-based API, EasyFJP also supports the Satin library, which provides SFJ-based parallelism, and the standard fork-join framework of Java [27]. To manually use Satin, methods considered for parallel execution must be indicated by the user through a *marker interface* that includes their exact signature and extends the *satin.Spawnable* interface. The class containing parallel (D&C) methods extends the *satin.SatinObject* class and implements the marker interface. In addition, the invocations to parallel methods are stored in local variables. After specifying parallel methods and inserting synchronization calls into the code, the developer must feed a compiled version of the application to the Satin compiler that translates, through Java bytecode instrumentation, each invocation to a parallel method into a Satin runtime task, so that forks are issued at runtime. Bytecode instrumentation means transforming the compiled version of a program to alter its semantics.

The purpose of the Satin builder is to automatically reproduce these tasks. The Satin builder generates the marker interface based on the operations specified within the configuration file of the application, and makes the peer extend and implement the required API classes and interfaces. Besides, the builder inserts appropriate calls to sync –the MFJ-based primitive of Satin– based on the output of the source code analysis of step 1.

Passing the source code of FibApp through the Satin builder (without taking into account policies) results in the following parallel code:

```
1  // Marker interface
2  public interface FibApp_Marker extends satin.Spawnable{
3   public long fibonacci(long n, ExecutionContext ctx);
4  }
5  // Peer
6  public class FibApp_Peer extends satin.SatinObject
7         implements FibApp_Marker, Serializable{
8   // Properties are copied as is from the original class
9   ...
10  public long fibonacci(long n){
11   return fib(n, initContext(...));
12  }
13  // The Satin-enabled method
14  public long fibonacci(long n, ExecutionContext ctx){
15   ...
16   f1 = fib(n - 1, updateContext(ctx, ...));
17   f2 = fib(n - 2, updateContext(ctx, ...));
18   super.sync();
19   return f1 + f2;
20  }
21  ...
22 }
```

The builder generates the FibApp_Marker marker interface (lines 1-4), and makes the peer implement it (line 7). MFJ-based synchronization in the resulting peer is managed via calls to the sync primitive of the Satin API (line 18).

4.3. Step 3: Policy declaration

Policies represent a mechanism to express, separately from the application logic, customized strategies to achieve better performance. Conceptually, a policy implements a user-specified rule that governs the behavior of an application within the underlying execution environment. As mentioned earlier, EasyFJP provides a policy-inspired tuning support that let developers introduce common FJP optimization heuristics without altering their applications by means of special Java classes.

Basically, policies represent tuning decisions that depend on the algorithmic nature of the application being parallelized. Particularly, policies model the notions of threshold, memoization and task placement, which are explained in the following subsections. These are, in other words, ways of regulating the levels of parallelism in an application for non-experienced developers.

Threshold-based policies. Threshold policies are used to avoid forking a subcomputation more than needed and otherwise execute it sequentially. For example, in the FibApp application, we may want to "throttle" the number of forks that are injected into the runtime system –and therefore the task granularity– depending on the depth

of the execution tree associated to the method at runtime. This decision is indicated to EasyFJP by associating the following policy to the fibonacci method:

```
class MyThresholdPolicy implements Policy{
 static final int THRESHOLD = 10;
 public boolean shouldFork(ExecutionContext ctx){
  return (ctx.getCurrentDepth() <= THRESHOLD);
 }
}
```

The code implements the Policy interface from the EasyFJP policy API and allows each execution of fibonacci to be forked provided the current depth of the execution tree of the method is below THRESHOLD. As mentioned in past paragraphs, ExecutionContext provides operations to further introspect the execution of the application, namely obtaining the values of method parameters. For example, for the recursive search method over an array of Figure 1, a policy may be associated to restrict parallelism depending on the size of the input array:

```
...
public boolean shouldFork(ExecutionContext ctx){
  int[] array = (int[])ctx.getArgument(1); // search(elem,array)
  return (array.length > MIN_ARRAY_SIZE);
}
...
```

The above policy code uses the ExecutionContext object to access the value of the second argument of each call to search to decide whether the size of the received array justifies another fork. To attach the above threshold policy to the FibApp class, we must supply the corresponding declaration in the configuration of the application.

Memoization policies. Memoization is another common optimization technique used to gain efficiency by having applications to avoid forking a subcomputation when results have been already computed. In this sense, in our FibApp class, we may want to avoid recalculating previously computed results, as the nature of the application makes subcomputations to overlap. 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:

```
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 EasyFJP to fork and hence to ignore previously computed results if the argument of a call to fibonacci is even. Moreover, whenever shouldFork evaluates to

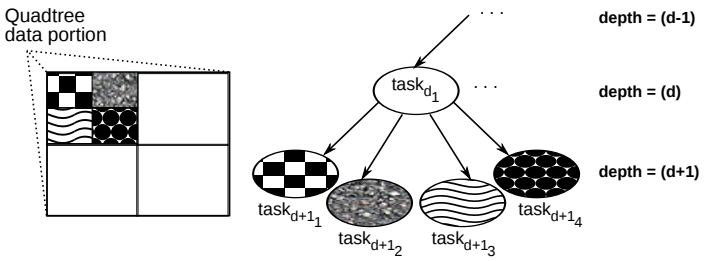


Fig. 2: An application processing a quadtree data structure.

false, EasyFJP 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 execution in parallel takes place. Depending on the target execution environment for the application (e.g., multi-core, cluster), memoization works by using a local in-memory or a distributed cache [27]. Memoization strategies like the one implemented by this policy, in which only a subset of already calculated results are reused, are useful in parallel optimization problems where forking for a subproblem may yield a better solution than reusing a similar computed suboptimal result [1].

Task placement policies. Task placement or mapping refers to the problem of assigning unfinished tasks to available executing nodes. This problem has been proved to be NP for both static as well as dynamic placement [22], i.e., when tasks are mapped in an off-line and a runtime fashion, respectively. Precisely, tasks resulting from executing D&C applications belong to the second category, because the execution of an individual task may trigger the execution of N more. Under EasyFJP, the node in charge of executing a task is not determined by the application but the underlying scheduler. However, the hierarchical task structure of EasyFJP applications indirectly determines task dependencies that, if ignored, may result in suboptimal performance. Then, 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. As such, every parallel task creates four more tasks, each in charge of processing a particular region of the data (see Figure 2). Now, let us suppose we execute this application on four clusters C_1, C_2, C_3 and C_4, connected through wide-area links. Assuming we are targeting a parallel library based on cluster-aware round robin scheduling such as the Satin framework and launching the execution of our application at cluster C_1, one possible task mapping is $task_{d_1}/task_{d+1_1} \rightarrow C_1$, $task_{d+1_2} \rightarrow C_2$, $task_{d+1_3} \rightarrow C_3$ and $task_{d+1_4} \rightarrow C_4$.

Alternatively, we could force $task_{d+1_1}$ and $task_{d+1_2}$ to be located at cluster C_1, and delegate the placement of the rest of the siblings tasks of $task_{d_1}$ to the scheduler (e.g., $task_{d+1_3} \rightarrow C_2$, $task_{d+1_4} \rightarrow C_3$). Depending on the amount of data interchanged between $task_{d_1}$ and $task_{d+1_1}/task_{d+1_2}$, the semi-automatic mapping may justify the loss of processing power at cluster C_4, to which no task would be assigned.

Roughly, this kind of decision can be specified in EasyFJP through a task placement policy, which tells, based on an API-exposed task identifier, where to submit parallel tasks for execution. Below is the policy code that implements the above mapping:

```
class MyTaskPlacementPolicy implements TaskPlacementPolicy{
 public boolean shouldMap(ExecutionContext ctx, TaskId id){
  if (ctx.getCurrentDepth() % 2 != 0){
   // This avoids overloading the local node
   return false;
  }
  return (id.getNumber() <= 2);
 }
 public InetAddress mapTo(ExecutionContext ctx, TaskId id){
  return InetAddress.getLocalHost();
 }
}
```

Basically, the shouldMap method decides whether to activate explicit task mapping for a given subcomputation, whereas mapTo indicates EasyFJP to which node the task should be submitted. Each task is assigned a unique identifier that comprises the depth associated to the task such as id.getDepth() plus one equals ctx.getCurrentDepth(), and a subsequent number. In this case, we have made the forked tasks to be placed in the same physical node as the parent tasks originating them. However, other complex actions could had been taken in this respect, 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 library API has support for explicit task mapping, a feature present in several Java-based parallel tools, e.g., ProActive and GridGain.

5. Experimental evaluation

The practical implications of using EasyFJP are determined by two essential aspects, i.e., how competitive is implicitly supporting FJP synchronization patterns in D&C codes compared to other parallel programming models, and whether policies are effective to tune parallelized applications. We have already conducted experiments to partially answer these questions in the context of the MFJ pattern with computational kernels and Grids [27], from which encouraging results were obtained. Next, we report experiments with the SFJ and MFJ synchronization patterns along with real-world applications, through our bindings to GridGain (Section 5.2) and Satin (Section 5.3) on both an emulated Grid and a cluster to better analyze the tradeoffs inherent to EasyFJP.

5.1. Testbeds and test applications

We set up a LAN comprising 8 dual core nodes with similar CPU capabilities running Ubuntu 11.04, Java 6, Satin 2.2, and GridGain 3.2.1. The nodes were connected through a 100 Mbps network. We refer to this environment as the *cluster*. Then, we established a wide-area Grid on top of this cluster by employing WANem version 2.2 [43], a software for emulating WAN conditions over a local area network. We emulated 3 remote clusters

C_1, C_2 and C_3 by using 2, 3 and 3 of the nodes of the cluster, respectively, which were connected together by using virtual Internet links. Each WAN link was a T1 connection (bandwidth of 1,544 Mbps) with a round-trip latency of 100 ms and a jitter of 5 ms, therefore inter-cluster latencies were in the range of 95-105 ms, which are network conditions found in Internet-wide Grids. The reason of using this setting was to provide a challenging testing scenario for EasyFJP. We refer to this second environment as the *emulated Grid* or simply *Grid*. In either environments, we configured middlewares in such a way 16 computing processors were available for the experiments.

Furthermore, we used two test applications. The first one was ray tracing, a widely-known rendering technique that generates a digital picture from an abstract description of a 3D scene [18]. Basically, we based our experiments on an existing D&C ray tracing code from the Satin project[5], which operates by deriving an initial image from an input scene, dividing this image to recursively apply the algorithm, and then joining the results. Computationally, the ray tracing application is both CPU and memory intensive.

The second application was local pairwise sequence alignment, a problem from Bioinformatics that involves representing a biological entity (e.g., a gene) in a computer-understandable way (e.g., strings) and manipulating this representation by using sequence alignment algorithms, which are usually computing intensive. Basically, we took an existing master-worker implementation of the application for aligning protein sequences whose code was obtained from the JPPF project [40]. This source code relies on JAligner [30], which given any pair of sequences outputs a coefficient representing the similarity level between these two by using a scoring matrix from a set of predefined matrixes. The original JPPF application aligned an unknown input sequence against an entire sequence database, which was replicated across the nodes of the experimental testbed to allow parallel tasks to locally access sequence data. The sequence alignment application makes extensive use of CPU and I/O, because of the many database accesses.

Moreover, various EasyFJP variants (SFJ-based and MFJ-based) of the ray tracing application were obtained by removing from the original Satin code any sentence related to parallelism or tuning application execution in order to derive its sequential D&C counterpart, and then automatically generating the corresponding parallel codes. The same was carried out to generate various EasyFJP variants of the sequence alignment application from the original JPPF code. The manual GridGain and Satin variants were obtained by directly altering the original parallel codes. We fed the applications with various 3D scenes, and real gene sequence databases from the NCBI (National Center for Biotechnology Information). For ray tracing, we used two scenes Scene 1 and Scene 2 with resolutions of 1024x1024 and 2048x2048. For sequence alignment, we compared five sequences against real protein sequence databases. Concretely, we employed five sequence databases of increasing sizes (4,289 up to 12,325 sequences).

5.2. SFJ-based parallelism

With respect to task granularity for ray tracing we used two task sizes: base and medium granularity. The former represents splitting the whole computation into the minimum

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

number of parallel tasks such that all processors (i.e., 16 in our case) get work to do, or in other words one task per processor. On the other hand, with the medium granularity, 64/256 tasks were generated when using the 1024x1024/2048x2048 image, which means 4/16 parallel tasks per node. For sequence alignment, we also employed two granularities, base (i.e., 1 parallel task per node), and medium, where the number of tasks to execute depended on the size of the input database for efficiency purposes. This is, the larger the database, the more the generated tasks, thus enabling for better parallelism. In this context, a task refers to a computing intensive calculation that actually processes a region of the input data. Therefore, the number of created tasks associated with each granularity was the number of launched workers in the manual GridGain variants, and the number of leaf nodes that resulted from executing the (recursive) EasyFJP variants. For either application, we used three EasyFJP variants:

- A variant including a default built-in policy that provides a base granularity for tasks. The policy finds the minimum factor f that satisfies $\#of\,tasks = fanout^f \geq \#of\,processors$, where fanout is the number of recursive calls included in the target D&C parallelized method. As suggested, this policy is attached by default to EasyFJP applications, thus no intervention from programmers is required.
- A variant using a threshold policy to configure medium task granularity.
- A variant using a policy that extended the above policy to place tasks processing near regions of the input data in the same cluster.
  Preliminary experiments regarding the use of another variant considering the default built-in policy plus task placement showed some serious performance problems. This is because the performance benefits of placing a set of related tasks in the same physical cluster scene becomes negligible when less tasks are executed. In other words, dynamically dividing the computation in few, computationally heavy tasks that are placed in the same physical cluster makes the rest of the clusters to be underused most of the time. Thus, we decided to left this variant out of the analysis.

Afterwards, we developed manual GridGain variants through its annotation-based parallel directives and its support for Google's MapReduce [11, 25]. We then modified the resulting codes to generate four manual GridGain variants by considering the two aforementioned granularities.

For the sake of fairness, all variants (automatic and manual) were configured to use the same load balancing mechanism at the platform (GridGain) level, namely round robin scheduling with the default configuration. According to the authors, this algorithm provides a fair distribution of tasks among the nodes of a Grid/cluster and works well in most cases. Basically, upon executing an application, the algorithm randomly picks a Grid node and then dynamically and sequentially assigns tasks for execution in a round-robin fashion. For the EasyFJP implementations using task placement policies, on the other hand, some of the generated tasks were heuristically and manually placed on nodes while for the rest this decision was delegated to this scheduler. In general, round-robin is known to be an algorithm that is much less effective when scheduling a *heavy-tailed* set of tasks [36], i.e., where processing few long-sized tasks takes a high percentage of the processing time for the entire set. However, all variants employed in the experiments generated tasks in charge of computing over similar portions of the input data, which made round-robin an effective scheduler for our test applications.

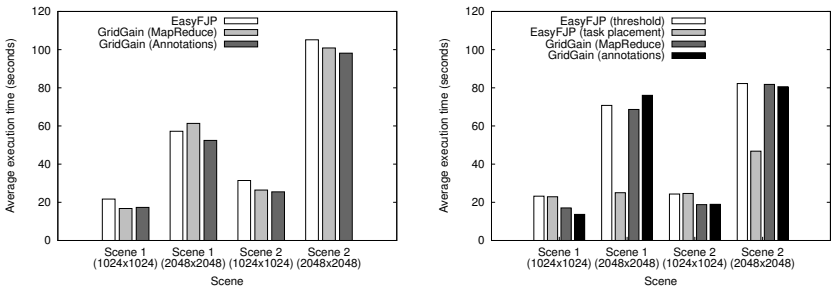
Emulated Grid. Figure 3 and Figure 4 illustrate the average execution time (AET) of the ray tracing and sequence alignment applications for 40 runs, respectively. In all cases, standard deviations were in the range of 5-12%. Note that this percentage is somewhat high, however it is mainly explained by the fact that GridGain, and hence the EasyFJP variants generated automatically, used a *random* load balancing support, plus the variability of the WAN links of the testbed in terms of latency.

For ray tracing, from Figure 3 (a), it can be seen that EasyFJP performed competitively for almost all scenes, because it consistently added an average overhead of around 5 seconds with respect to the manual variants. The exception to this rule was the combination GridGain (MapReduce) variant and Scene 1 (1024x1024), which showed an overhead in favor of EasyFJP, but this may be explained by the unpredictable nature of the execution environment. We believe these results are acceptable taking into account that EasyFJP automates SFJ-based parallelism, and the non-invasive nature of policies allowed the EasyFJP code to remain clean from these kind of rules, which simplified the implementation of the medium-grained variants. Precisely, by observing Figure 3 (b), some very interesting facts arise. For the case of the 1024x1024 resolution scenes, again, EasyFJP added an overhead of few seconds with regard to its competitors. However, for the case of the 2048x2048 scenes, using a raw threshold allowed EasyFJP to perform even closer to the two manual variants, and task placement introduced gains of up to 63% with respect to the most efficient manual variants in either cases. It is worth mentioning that the weak point of implementing effective task placement policies is the requirement of a deeper knowledge on the EasyFJP API. Users without the necessary knowledge can fall back to threshold policies, which are much easier to specify.

For sequence alignment, the execution times uniformly increased as database sizes increased for all tests, which shows a good overall correlation of the different variants. Interestingly, for the base granularity (see Figure 4 (a)) and databases DB 2, DB 3 and DB 5, EasyFJP performed better than the two manual variants. For DB 1 and DB 4, on the other hand, EasyFJP added a very small performance overhead. Although this is interesting, our goal is not to outperform existing Grid libraries but simplifying their usage while achieving competitive performance. Note that Figure 4 (b) reveals similar results when relying on medium-grained parallel tasks.

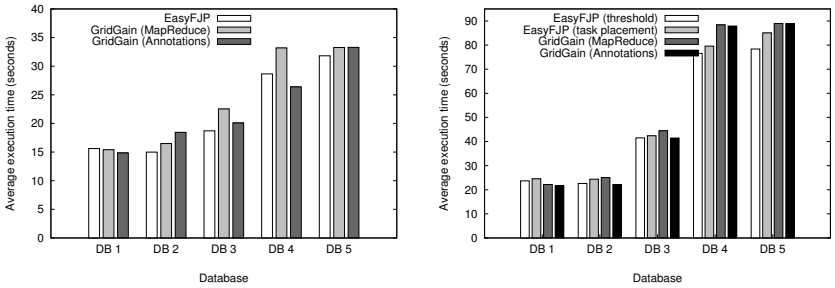
Besides, from Figure 4 (b), it can be seen that task placement did not help in reducing execution times since, unlike ray tracing, parent and child parallel tasks did not interchange large amounts of data. In other words, for sequence alignment, we observed that manually mapping individual tasks to nodes naturally lead to less communication delays but also to much lower levels of processor usage. This does not imply that task placement policies are not effective but their usage should be decided depending on the nature of the application, otherwise they may yield negative results. Last but not least, we emphasize the positive aspect of policies given by the fact that whenever a tuning decision does not work as expected, the programmer can easily switch between several policies with no harm to the logic of its application.

Cluster. Figure 5 and Figure 6 illustrate the average execution time (AET) of the ray tracing and sequence alignment applications for 25 runs, respectively. We performed less runs compared to the emulated Grid as the standard deviations were much lower.



(a) Base task granularity (b) Medium task granularity

Fig. 3: SFJ-based EasyFJP and manual GridGain variants: AET on the emulated Grid (ray tracing)

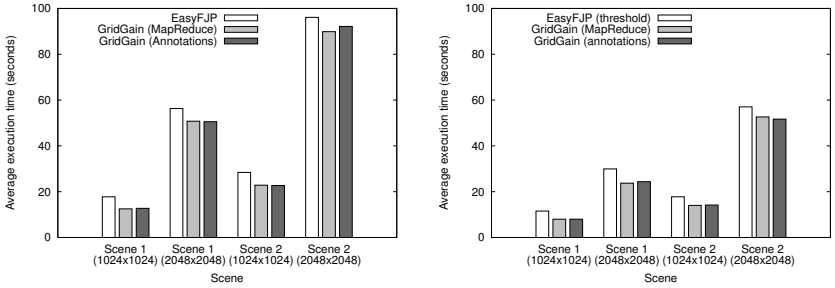


(a) Base task granularity (b) Medium task granularity

Fig. 4: SFJ-based EasyFJP and manual GridGain variants: AET on the emulated Grid (sequence alignment)

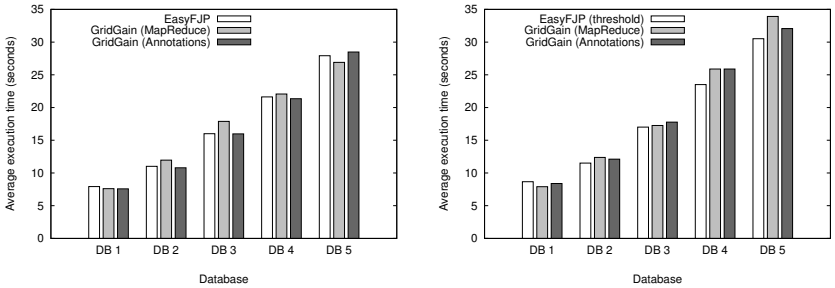
Moreover, unlike the previous subsection, it can be seen that the variants using the task placement policy were not used as this policy does not apply to this execution environment.

As illustrated in Figure 5, EasyFJP incurred an average and constant overhead of around 6 seconds with respect to its competitors. As suggested by Figure 6, this problem was not present with the sequence alignment application. A main component of this overhead is caused by the policy support of EasyFJP. Particularly, for ray tracing, the shouldFork(ExecutionContext ctx) method of the policies of both EasyFJP variants receives a copy of the parameters of the parallelized method the policies are attached to. Therefore, upon each policy invocation, the ctx object is populated with a copy of the subimage being processed plus some image meta-data (dimensions, color information, etc.) that were present in the signature of the parallelized method. However, the implemented threshold policy, upon deciding whether to fork a task into more subtasks



(a) Base task granularity (b) Medium task granularity

Fig. 5: SFJ-based EasyFJP and manual GridGain variants: AET on the cluster (ray tracing)



(a) Base task granularity (b) Medium task granularity

Fig. 6: SFJ-based EasyFJP and manual GridGain variants: AET on the cluster (sequence alignment)

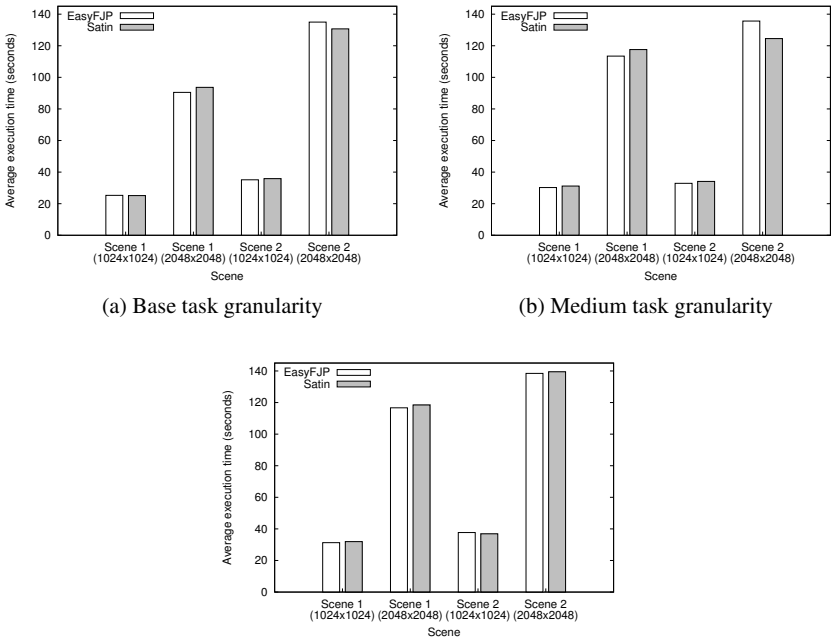
or not, it only needs the subimage dimensions. In this sense, lots of objects are unnecessarily created/copied. Then, to avoid this overhead, one solution could be parametrizing policies with the minimal set of parameters. We are extending the way developers can currently specify policies at the Step 3 of the process of Section 4 by allowing them to build a mapping between a target method's parameters and the input data actually used by the associated policy. Consequently, at runtime, EasyFJP could perform only the copy operations specified in the mappings.

5.3. MFJ-based parallelism

In these experiments, we compared the performance of the applications generated automatically via EasyFJP and our bindings to Satin versus the ones manually coded by using the Satin API. The next paragraphs report the obtained results.

Emulated Grid. For both applications, we also employed base and medium task granularities. Additionally, unlike GridGain, Satin is designed to support finer forms of parallelism. Due to this particularity of the targeted parallel library, we also used fine task granularities, comprising twice as many runtime tasks compared to the medium granularity. Furthermore, for controlling task granularity in the EasyFJP variants, threshold-based policies were also employed.

Figures 7 and 8 illustrate the AET of the MFJ-based ray tracing and sequence alignment codes for 40 runs. Again, standard deviations were in the range of 8-17%, which was due to the variability of the WAN links of the testbed, plus the fact that Satin and hence EasyFJP exploited the CRS [44] task scheduling algorithm of Satin, which implements a cluster-aware *random* task stealing mechanism. With CRS, when a Grid node becomes idle, it attempts to steal an unfinished task both from nodes belonging to the same local cluster or external nodes, however intra-cluster steals have higher priority than inter-cluster ones, minimizing expensive WAN communication.



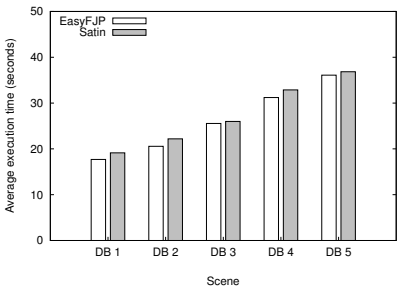
(a) Base task granularity

(b) Medium task granularity

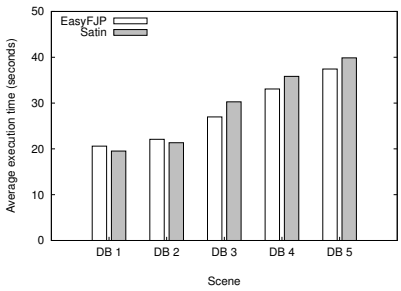
(c) Fine task granularity

Fig. 7: MFJ-based EasyFJP and Satin variants: AET on the emulated Grid (ray tracing)

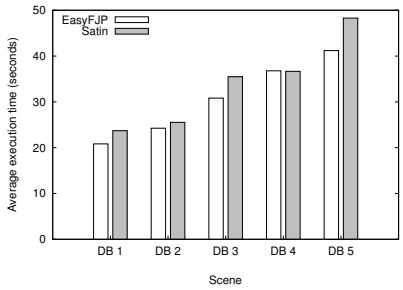
All in all, the codes performed very well compared to Satin, considering that our goal is not to outperform existing Grid libraries but automating as much as possible their



(a) Base task granularity



(b) Medium task granularity



(c) Fine task granularity

Fig. 8: MFJ-based EasyFJP and Satin variants: AET on the emulated Grid (sequence alignment)

usage while achieving competitive performance. From Figure 7 (a) it can be seen that EasyFJP either performed very closely with respect to Satin or incurred in overheads of few seconds. Specifically, the most visible execution overhead was only for the case of base and medium granularities and Scene 2 (2048x2048), however, with the threshold policy using the fine granularity, EasyFJP slightly outperformed Satin whereas remained competitive for the rest of the scenes and resolutions (see Figure 7 (c)). In practice, this means that whenever a desired performance level is not reached for an experimental scenario, another policy can be non-invasively configured to the same application code. In general, this is more difficult to achieve manually with Satin, since the code controlling granularity may be scattered across the application logic.

For sequence alignment and the medium task granularity, EasyFJP incurred in overheads of 5% and 3% for DB1 and DB2, respectively. Furthermore, for the fine task granularity and DB4, EasyFJP had a negligible overhead. In general, EasyFJP outperformed Satin in 12 out of 15 granularity-input combinations, with gains of up to 14%. In principle, this may seem confusing since the generated EasyFJP code uses Satin as the underlying support for execution. This is explained in part by the random nature of the Satin scheduler, but the main reason is that the code executed by the Satin run-

time in either cases is subject to different computational requirements. The pure Satin versions of the applications were parallelized by hand, while the EasyFJP counterparts were parallelized automatically and as such they contained more sentences for supporting policies. Since the sequence alignment is much less CPU intensive compared to the ray tracing application, having more sentences means heavier tasks, which favors scheduling for libraries designed to efficiently handle CPU-bound tasks like Satin.

Standard deviations were 9-15% for the case of Satin but in the range 8-17% for the case of EasyFJP. This fact may suggest that the execution time of the EasyFJP variants of sequence alignment was more affected by the data-intensive nature of the application, however this should be further corroborated. To sum up, we believe these are acceptable results that justify the use of EasyFJP to ease the exploitation of the Satin library in particular –via the MFJ synchronization pattern– in similar Grid scenarios.

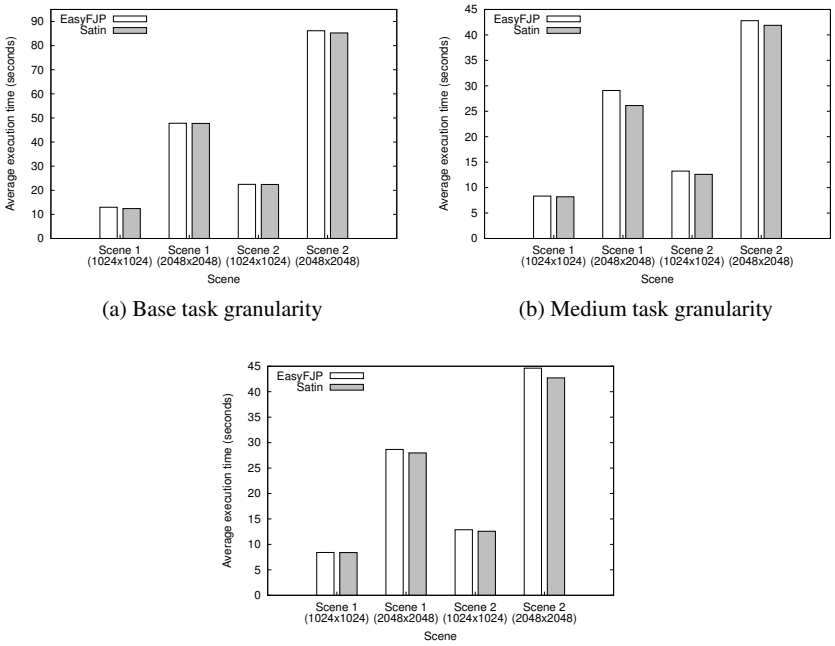
Cluster. In a final round of experiments, we compared the performance of EasyFJP and Satin in a computer cluster. To schedule parallel tasks, we used the random job stealing algorithm (or RS for short) provided by Satin. It is worth noting that the "workhorse" of Satin for running tasks is the CRS algorithm, however we also considered a cluster and therefore RS for the sake of completeness.

Figures 9 and 10 show the resulting times for the ray tracing and the sequence alignment applications. For the case of ray tracing, we did not obtain significant differences in favor of any of the two approaches, and consequently we can say that in general EasyFJP performed competitively. For sequence alignment and base granularity, however, EasyFJP performed worse. The reader should recall that our synchronization algorithms are heuristics that do not guarantee optimal barrier placement compared to programs parallelized by hand in a smart way. Then, when combined with the default task granularity policy, results may be suboptimal. It can be seen however that the use of threshold policies (particularly the variant shown in Figure 10 (c)) ameliorated the overheads, which enforces the experimental findings from the previous subsections.

6. Conclusions

In this paper we have described EasyFJP, an approach to simplify the parallelization of divide and conquer sequential Java applications. EasyFJP introduces the concept of FJP synchronization pattern, i.e., common forms of fork-join parallelism present in many task-based Java frameworks and libraries. EasyFJP exploits this through semi-automatic algorithms and source code rewriting techniques to generate parallel applications. Also, execution of parallel applications is performed by leveraging the scheduling and load balancing services of existing parallel tools, and can be explicitly adjusted for optimization purposes via *policies*.

We have shown that this approach has the potentiality to offer a better balance to the "ease of use and versatility versus performance" tradeoff inherent to tools for parallel programming, plus the flexibility of generating code to exploit various parallel libraries and FJP optimization heuristics. The experimental results obtained based on real world applications and execution environments, in conjunction with the ones preliminary reported in [27] confirm that implicit synchronization based on FJP patterns and policy--



(a) Base task granularity

(b) Medium task granularity

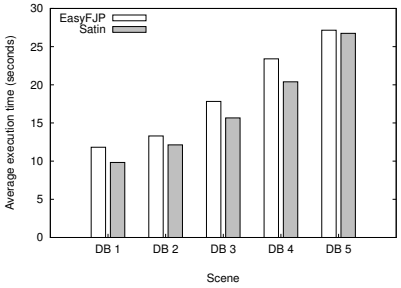
(c) Fine task granularity

Fig. 9: MFJ-based EasyFJP and Satin variants: AET on the cluster (ray tracing)

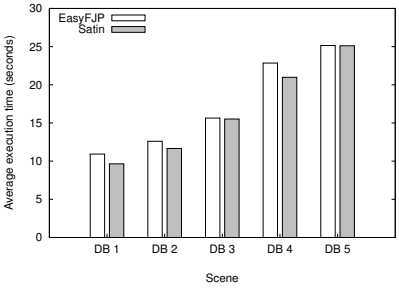
oriented explicit tuning, glued together through generative programming, are a viable approach to PaaC from a practical perspective. Interestingly, we have shown that using EasyFJP and targeting such libraries –in this case GridGain and Satin– does not lead to resigning performance compared to manually using the libraries.

Up to now, EasyFJP deals with two broad parallel *concerns*, namely (pattern-based) task synchronization and application tuning. We are investigating how to incorporate to our approach other common concerns in parallel programming and specially FJP applications such as inter-task communication, and adapting our ideas to newer parallel environments such as Cloud environments [6, 34], which have recently gained much attention and have became a hot topic in high performance computing. However, this is a mid-term research goal that should be deeply studied.

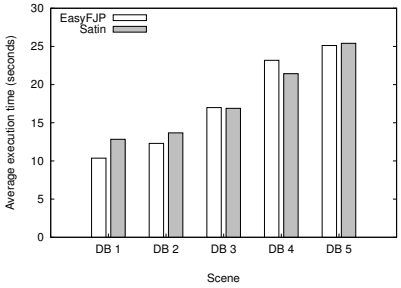
Moreover, there is a recent towards programming tools that simplify parallel software development. One of the aims of these tools is reducing the analysis and transformation burden when parallelizing sequential programs, which improves programmer productivity [12]. In this line, we are building IDE support to simplify the adoption and use of EasyFJP. As a starting point, we will adopt Eclipse, which is very popular among Java developers. Finally, we have produced a materialization of our ideas to support the development of parallel applications within pure engineering communities, where



(a) Base task granularity



(b) Medium task granularity



(c) Fine task granularity

Fig. 10: MFJ-based EasyFJP and Satin variants: AET on the cluster (sequence alignment)

scripting languages such as Python and Groovy are the common choice [31, 33, 32]. At present, we have redesigned the policy support of EasyFJP to allows developers to code policies in Java as well as Python and Groovy. We also plan to materialize EasyFJP concepts into these scripting languages. Then, we will investigate how to port and exploit the parallelization heuristics of EasyFJP apart from its policy mechanism, which will require determining which is the most appropriate fork-join parallelization pattern for these languages.

References

1. Alba, E., Blum, C., Asasi, P., Leon, C., Gomez, J.A.: Optimization Techniques for Solving Complex Problems. Parallel and Distributed Computing, Wiley Publishing (Mar 2009)
2. Aldinucci, M., Danelutto, M., Dazzi, P.: Muskel: An expandable skeleton environment. Scalable Computing: Practice and Experience 8(4), 325–341 (2007)
3. Armstrong, J.: Programming Erlang: Software for a Concurrent World. Pragmatic Bookshelf (2007)

4. Baduel, L., Baude, F., Caromel, D., Contes, A., Huet, F., Morel, M., Quilici, R.: Grid Computing: Software Environments and Tools, chap. Programming, Composing, Deploying on the Grid, pp. 205–229. Springer, Berlin, Heidelberg, and New York (2006)
5. Bull, J., Kambites, M.: JOMP-an OpenMP-like interface for Java. In: ACM Conference on Java Grande (JAVA '00), San Francisco, CA, USA. pp. 44–53. ACM Press, New York, NY, USA (2000)
6. Buyya, R.: Market-oriented Cloud Computing: Vision, hype, and reality of delivering computing as the 5th utility. In: 9th IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGRID '09). p. 1. IEEE Computer Society, Washington, DC, USA (2009)
7. Caminero, A., Rana, O., Caminero, B., Carrión, C.: Network-aware heuristics for inter-domain meta-scheduling in Grids. Journal of Computer and System Sciences 77(2), 262–281 (2011)
8. Chan, H.N., Gallagher, A., Goundan, A., Au Yeung, Y.L., Keen, A., Olsson, R.: Generic operations and capabilities in the JR concurrent programming language. Computer Languages, Systems & Structures 35(3), 293–305 (2009)
9. Chandra, R., Dagum, L., Kohr, D., Maydan, D., McDonald, J., Menon, R.: Parallel Programming in OpenMP. Morgan-Kaufmann Publishers Inc., San Francisco, CA, USA (2000)
10. Danaher, J., Lee, I., Leiserson, C.: Programming with exceptions in JCilk. Science of Computer Programming 63(2), 147–171 (2006)
11. Dean, J., Ghemawat, S.: MapReduce: A flexible data processing tool. Communications of the ACM 53, 72–77 (2010)
12. Dig, D.: A refactoring approach to parallelism. IEEE Software 28(1), 17–22 (2011)
13. Duarte, R., Mota, A., Sampaio, A.: Introducing concurrency in sequential Java via laws. Information Processing Letters 111(3), 129–134 (2011)
14. Foster, I.: The Grid: Computing without bounds. Scientific American 288(4), 78–85 (Apr 2003)
15. Freeh, V.: A comparison of implicit and explicit parallel programming. Journal of Parallel and Distributed Computing 34(1), 50–65 (1996)
16. GridGain Systems: GridGain = Real Time Big Data. `http://www.gridgain.com` (2011)
17. Haustein, M., Lohr, K.P.: JAC: Declarative Java concurrency. Concurrency and Computation: Practice and Experience 18(5), 519–546 (2006)
18. Heckbert, P., Haines, E.: A ray tracing bibliography. In: Glassner, A. (ed.) Introduction to Ray Tracing, pp. 295–303. Academic Press, Inc. (1989)
19. Hudak, P., Hughes, J., Jones, S., Wadler, P.: A history of Haskell: Being lazy with class. In: 3rd. ACM SIGPLAN Conference on History of Programming Languages (HOPL III), San Diego, California, USA. pp. 1–55. ACM Press, New York, NY, USA (2007)
20. Johnson, R.: J2EE development frameworks. Computer 38(1), 107–110 (2005)
21. Jugravu, A., Fahringer, T.: JavaSymphony, a programming model for the Grid. Future Generation Computer Systems 21(1), 239–246 (2005)
22. Kim, J.K., Shivle, S., Siegel, H., Maciejewski, A., Braun, T., Schneider, M., Tideman, S., Chitta, R., Dilmaghani, R., Joshi, R., Kaul, A., Sharma, A., Sripada, S., Vangari, P., Yellampalli, S.: Dynamically mapping tasks with priorities and multiple deadlines in a heterogeneous environment. Journal of Parallel and Distributed Computing 67(2), 154–169 (2007)
23. Lea, D.: The java.util.concurrent synchronizer framework. Science of Computer Programming 58(3), 293–309 (2005)
24. Lee, E.: The problem with threads. Computer 39(5), 33–42 (2006)
25. Marozzo, F., Talia, D., Trunfio, P.: P2P-MapReduce: Parallel data processing in dynamic Cloud environments. Journal of Computer and System Sciences 78(5), 1382–1402 (2012)
26. Mateos, C., Zunino, A., Campo, M.: JGRIM: An approach for easy gridification of applications. Future Generation Computer Systems 24(2), 99–118 (2008)

27. Mateos, C., Zunino, A., Campo, M.: An approach for non-intrusively adding malleable fork/join parallelism into ordinary JavaBean compliant applications. Computer Languages, Systems & Structures 36(3), 288–315 (2010)
28. Mateos, C., Zunino, A., Campo, M.: On the evaluation of gridification effort and runtime aspects of JGRIM applications. Future Generation Computer Systems 26(6), 797–819 (2010)
29. McDowell, C., Helmbold, D.: Debugging concurrent programs. ACM Computing Surveys 21(4), 593–622 (1989)
30. Moustafa, A.: JAligner: Open source Java implementation of Smith-Waterman. http://jaligner.sourceforge.net (2005)
31. Oliphant, T.: Python for scientific computing. Computing in Science and Engineering 9(3), 10–20 (2007)
32. Papadimitriou, S., Terzidis, K., Mavroudi, S., Likothanassis, S.: Scientific scripting for the Java platform with jLab. Computing in Science and Engineering 11, 50–60 (2009)
33. Pérez, F., Granger, B., Hunter, J.: Python: An ecosystem for scientific computing. Computing in Science and Engineering 13(2), 13–21 (2011)
34. RuWei, H., XiaoLin, G., Wei, Y.S.Z.: Study of privacy-preserving framework for Cloud storage. Computer Science and Information Systems 8(3), 801–819 (2011)
35. Seco, J.C., Silva, R., Piriquito, M.: Component J: A component-based programming language with dynamic reconfiguration. Computer Science and Information Systems 5(2), 63–86 (2008)
36. Semchedine, F., Bouallouche-Medjkoune, L., Aissani, D.: Task assignment policies in distributed server systems: A survey. Journal of Network and Computer Applications 34(4), 1123–1130 (2011)
37. Shafi, A., Carpenter, B., Baker, M., Hussain, A.: A comparative study of Java and C performance in two large-scale parallel applications. Concurrency and Computation: Practice and Experience 21(15), 1882–1906 (2009)
38. Shang, Y., Lu, G., Shang, L., Wang, G.: Parallel processing on block-based gauss-jordan algorithm for desktop Grid. Computer Science and Information Systems 8(3), 739–759 (2011)
39. Sobral, J., Proença, A.: Enabling JaSkel skeletons for clusters and computational Grids. In: International Conference on Cluster Computing (CLUSTER 2007), Austin, Texas, USA. pp. 365–371. IEEE Computer Society (2007)
40. Sourceforge.net: Java Parallel Processing Framework. http://www.jppf.org (2009)
41. Sun Microsystems: JavaBeans Specification. http://java.sun.com/javase/technologies/desktop/javabeans/docs/spec.html (2010)
42. Taboada, G.L., Ramos, S., Expósito, R.R., Touriño, J., Doallo, R.: Java in the high performance computing arena: Research, practice and experience. Science of Computer Programming (2011), to appear
43. TATA Consultancy Services: The Wide Area Network Emulator. http://wanem.sourceforge.net (2009)
44. Wrzesinska, G., van Nieuwport, R., Maassen, J., Kielmann, T., Bal, H.: Fault-tolerant scheduling of fine-grained tasks in Grid environments. International Journal of High Performance Computing Applications 20(1), 103–114 (2006)
45. Wyatt, B., Kavi, K., Hufnagel, S.: Parallelism in object-oriented languages: A survey. IEEE Software 9(6), 56–66 (1992)
46. Zhang, B.Y., Mo, Z.Y., Yang, G.W., Zheng, W.M.: Dynamic load-balancing and high performance communication in JCluster. 21th IEEE International Parallel and Distributed Processing Symposium (IPDPS '07), Long Beach, California, USA p. 227 (2007)
47. Zhang, H., Lee, J., Guha, R.: VCluster: A thread-based Java middleware for SMP and heterogeneous clusters with thread migration support. Software: Practice and Experience 38(10), 1049–1071 (2008)