# A semi-automatic parallelization tool for Java based on fork-join synchronization patterns

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

[1] ISISTAN Research Institute - Also CONICET
[2] UNICEN University

**Abstract.** Because of the increasing availability of multi-core machines, clusters, Grids, and combinations of these, there is now plenty of computational power. However, today's programmers are not fully prepared to exploit distribution and parallelism. In this sense, the Java language has helped in handling the heterogeneity of such environments, but there is a lack of facilities to easily distributing and parallelizing applications. One solution to mitigate this problem seems to be the synthesis of semi-automatic parallelism and Parallelism as a Concern (PaaC), which promotes parallelizing applications along with as little modifications on sequential codes as possible. In this paper, we discuss a new approach that aims at overcoming the drawbacks of current Java-based parallel and distributed development tools.

**Keywords:** Parallel software development, distributed computing, fork-join synchronization patterns, Java, EasyFJP

## 1 Introduction and problem statement

The rise of powerful execution environments doubtlessly calls for new parallel and distributed programming tools. Many existing tools remain hard to use for non-experienced programmers, and prioritize performance over other important attributes such as code invasiveness and execution environment independence. Simple parallel programming models are essential for helping "sequential" developers to gradually move into the mainstream. Low code invasiveness and environment neutrality are also important since they allow for hiding parallelism and distribution from applications.

In dealing with the software diversity of such environments –specially distributed ones– Java is interesting as it offers platform independence and competitive performance compared to conventional languages. However, most Java tools have focused on running on *one* environment. Besides, they often offer developers APIs for programmatically coordinating subcomputations. This needs knowledge on parallel/distributed programming, and output codes are tied to the library employed, compromising code maintainability and portability to other libraries. All in all, parallel programming is nowadays the rule and not the exception. Hence, researchers and software vendors have put on their agenda the long-expected goal of versatile parallel tools delivering minimum development effort and code intrusiveness.

To date, several Java tools for scaling CPU-hungry applications have been proposed. Regarding multicore programming, Doug Lea's framework [1] and JCilk [2] extend the Java runtime library with concurrency primitives. Alternatively, JAC [3] separates application logic from thread management via annotations. Duarte et al. [4] address the same goal by automatically deriving thread-enabled codes from sequential ones based on algebraic laws. Regarding cluster/Grid programming, most tools offer APIs to manually create and coordinate parallel computations (e.g. JavaSymphony [5], JCluster [6], JR [7], VCluster [8] and Satin [9]). A distinctive feature of them compared to other Java libraries for building classical master-worker applications such as GridGain [10] or JPPF [11] is that the former group supports complex parallel applications structures. All in all, tools in both groups are designed for programming parallel codes rather than transforming ordinary codes to cluster and Grid-aware ones.

Regardless the environment, parallel programming can be classified into implicit and explicit [12]. The former allows programmers to write applications without thinking about parallelism, which is automatically performed by the runtime system, but performance may be suboptimal. Explicit parallelism supplies APIs so that developers have more control over parallel execution to implement efficient applications, but the burden of managing parallelism falls on them. Although designed with simplicity in mind, most efforts are still inspired by explicit parallelism. Parallelizing applications requires learning parallel programming APIs. From a software engineering standpoint, parallelized codes are hard to maintain and port to other libraries. In addition, these approaches lead to source code that contains not only statements for managing subcomputations but also for tuning applications. This makes such tuning logic obsolete when an application is ported for example from a cluster to a Grid.

An alternative approach to traditional explicit parallelism is to treat parallelism as a *concern*, thus avoiding mixing application logic with code implementing parallel behavior (Table 1). This has gained momentum as reflected by Java tools that partly or entirely rely on mechanisms for separation of concerns, e.g. code annotations (JAC [3]), metaobjects (ProActive [13]) and Dependency Injection (JGRIM [14]). Other efforts support the same idea through AOP, and *skeletons*, which capture recurring parallel programming patterns such as pipes and heartbeats in an application-agnostic way. Skeletons are instantiated by wrapping sequential codes or specializing framework classes, as in [15,16].

Current approaches pursuing PaaC fall short with respect to applicability, code intrusiveness and expertise. Tools designed to exploit single machines are usually not applicable to clusters/Grids, and approaches designed to exploit

| | **Implicit parallelism** | **Explicit parallelism** | |
|---|---|---|---|
| | | **Parallelism as a Concern (PaaC)** | **Invasive parallelism** |
| Is the programmer aware of parallelism? | NO | YES | YES |
| Is the source code of the sequential application manually modified to introduce parallelism? | NO | NO (or aiming to) | YES |
| Examples | Languages such as High Performance Fortran, Microsoft's Axum, MATLAB M-code, etc. | - Code Annotations<br>- Metaobjects<br>- Dependency Injection<br>- AOP<br>- Non-invasive Skeletons<br>- **Generative programming** | - API functions<br>- Method-level compiler directives |

**Table 1.** Parallelism in Java: Taxonomy (adapted from [17])

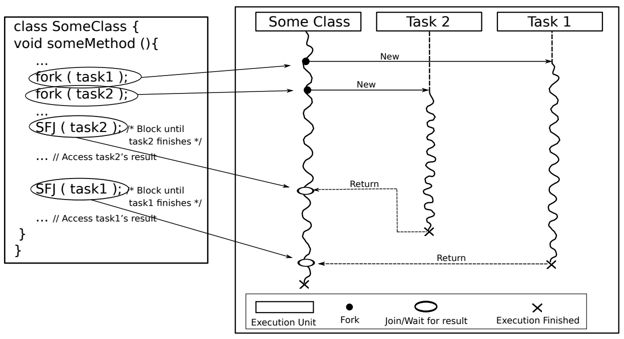
these settings incur in overheads when used in multicore machines. Moreover, approaches based on annotations require explicit modifications to insert parallelism and application-specific optimizations that obscure final codes. Metaobjects and specially AOP cope with this problem, but at the expense of incepting another programming paradigm. Lastly, tools providing support for various parallel patterns feature good applicability in respect to the variety of applications that can be parallelized, but require solid knowledge on parallel programming.

We propose EasyFJP, a tool aimed at unexperienced developers that offers means for parallelizing sequential applications. EasyFJP exploits PaaC by adopting a base programming model providing opportunities for enabling *implicit* nevertheless versatile forms of parallelism. EasyFJP also employs generative programming to build code that leverages existing parallel libraries for various environments. Developers proficient in parallel programming can further optimize generated codes via an *explicit*, but non-invasive tuning framework. EasyFJP is an ongoing project for which encouraging results in the context of the Satin library has been obtained [17]. In this paper, we show the various extensions and adaptations to EasyFJP in order to support another class of libraries in general and the well-known GridGain library in particular.

The paper is organized as follows. Section 2 introduces the concept of fork-join parallelism. Then, Section 3 overviews the EasyFJP project and its main technical aspects. In Section 4 an implementation of EasyFJP is explained in detail. En empirical validation of EasyFJP implementation with several variants is reported in Section 5. Finally, Section 6 concludes the paper and presents some future research works.

## 2 Fork-join parallelism: Basic concepts

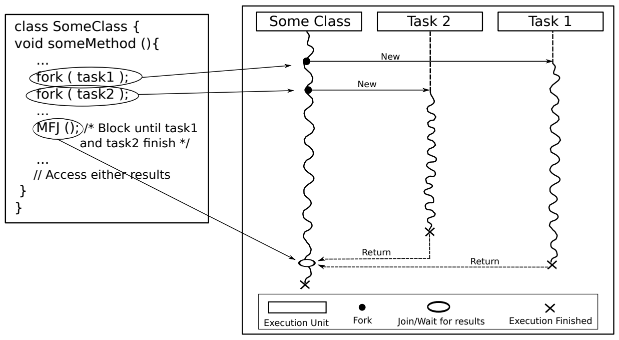
Fork-join parallelism (FJP) is a simple but effective technique that expresses parallelism via two primitives: *fork*, which starts the execution of a method in parallel, and *join*, which blocks a caller until the execution of methods finishes. FJP represents an alternative to threads, which have received criticism due to their inherent complexity. In fact, Java, which has offered threads as first-class citizens for years, includes now an FJP framework for multicore CPUs, which is based on Doug Lea's work.



```
class SomeClass {
void someMethod (){
  ...
  fork ( task1 );
  fork ( task2 );
  ...
  SFJ ( task2 ); /* Block until
                 task2 finishes */
  ... // Access task2's result

  SFJ ( task1 ); /* Block until
                 task1 finishes */
  ... // Access task1's result
 }
}
```

**Fig. 1.** Simple Fork-Join synchronization pattern



```
class SomeClass {
void someMethod (){
  ...
  fork ( task1 );
  fork ( task2 );
  ...
  MFJ (); /* Block until task1
          and task2 finish */
  ...
  // Access either results
 }
}
```

**Fig. 2.** Multiple Fork-Join synchronization pattern

FJP is not circumscribed to multicore programming, but is also applicable in execution environments where the notions of "tasks" and "processors" exist. For instance, forked tasks can be run on a cluster. Recently, Computational Grids, which arrange resources from geographically dispersed sites, have emerged as another environment for parallel computing. Then, 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 by using environment-specific execution platforms.

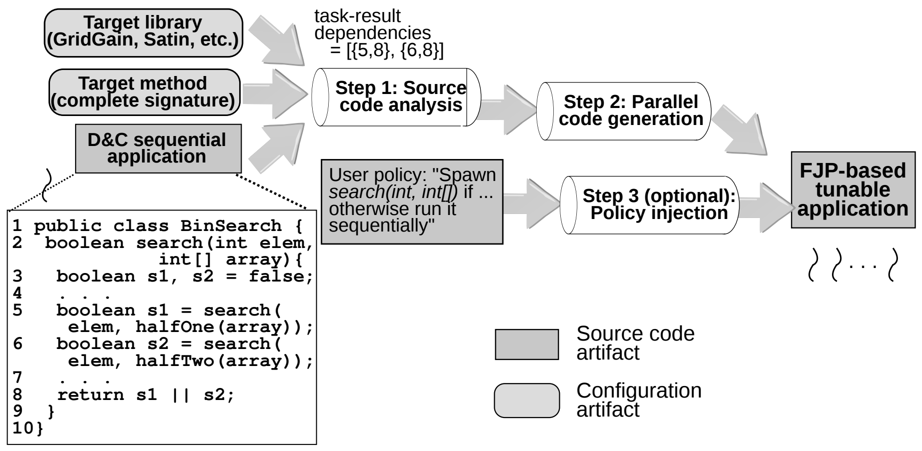
Broadly, current Java parallel libraries relying on task-oriented execution models offer API primitives to fork one or many tasks simultaneously, which are firstly mapped to library-level execution units. There are, however, operational differences among libraries concerning the primitives to synchronize subcomputations. We have observed that there are two *FJP synchronization patterns*: single-fork join (SFJ) and multi-fork join (MFJ). The former represents one-to-one relationships between fork and join points: a programmer must block its application to wait for each task result. With MFJ, the programmer waits for the results of the tasks launched up to a synchronization call. In the following codes, two SFJ calls are necessary to safely access the results of $task_1$ and $task_2$ (Fig. 1), whereas the same behavior is achieved with one MFJ call (Fig. 2).

Examples of Java parallel libraries and their support for these patterns are Satin (MFJ), ProActive (SFJ, MFJ), GridGain (SFJ) and JPPF (SFJ), which developers take advantage of through API calls. As discussed, this requires to learn an API, and ties the code to the library at hand. Even more important, managing synchronism for real-world applications is error prone and time-consuming.

## 3 The EasyFJP project: FJP as a concern

Intuitively, FJP is suitable for parallelizing divide and conquer (D&C) applications. This is because there is a direct association between Fork and Join points with recursive invocations and the use of recursive results respectively. For instance, Binary Search D&C algorithm (Fig. 3) that serves as input of the parallellization process, has two recursive calls or Fork points (lines 5 and 6) and two access to recursive results or Join points (line 8). The EasyFJP project [17] goals is to design source code analysis algorithms and code generation techniques to inject SFJ and MFJ into sequential D&C codes. EasyFJP includes a semi-automatic process (Fig. 3) that automatically outputs library-dependent parallel codes with hooks for attaching user optimizations.

First, at **step 1**, given a sequential application, a target D&C method of this application and a target parallel library as input, EasyFJP performs an analysis of the source code to spot the points that perform recursive calls and access to recursive results. As a convention to facilitate the analysis it is important that programmers write the sequential application assigning the results of recur-


Target library (GridGain, Satin, etc.)
Target method (complete signature)
D&C sequential application
task-result dependencies = [{5,8}, {6,8}]
Step 1: Source code analysis
Step 2: Parallel code generation
User policy: "Spawn search(int, int[]) if ... otherwise run it sequentially"
Step 3 (optional): Policy injection
FJP-based tunable application
Source code artifact
Configuration artifact

```
1 public class BinSearch {
2  boolean search(int elem,
             int[] array){
3   boolean s1, s2 = false;
4   . . .
5   boolean s1 = search(
     elem, halfOne(array));
6   boolean s2 = search(
     elem, halfTwo(array));
7   . . .
8   return s1 || s2;
9  }
10}
```

**Fig. 3.** EasyFJP: Parallelization process

sive calls to local variables. Depending on the target parallel library selected, EasyFJP uses an MFJ or a SFJ-inspired algorithm to detect prospective fork and join points, but the algorithms themselves do not depend on the parallel library selected. For brevity, below we discuss the SFJ algorithm; [17] presents its MFJ counterpart. As such, the fork-join pattern supported by this algorithm represents the main difference between this work and [17].

The SJF-based algorithm (see Alg. 1 and Table 2) works by depth-first walking the instructions and detecting where a local variable is *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, whereas it is used when its value is read. As input, the algorithm operates on a tree derived from the target method source code. Nodes in this tree are method scopes, while ancestor-descendant relationships represent nested scopes. First, the procedure IdentifyForkPoints search for local variables placed on the left side of an assignment operation where the right side is a recursive call. These variables would be parallel variables and the recursive calls are Fork points. The list of Fork points is passed as argument to the procedure IdentifyJoinPoints which, for every fork point, examines the sentences looking for every use of the result of the parallel variable associated to a Fork point. All the resulting ocurrences are marked as Join points of the current Fork point. Finally, the algorithm passes on to step 2 the list of recursive call and its corresponding uses of recursive results so that it can transform them into API calls.

**At step 2,** based on previous identified recursive calls and uses of recursive results, EasyFJP modifies the source code to call a library-specific fork and join primitive between the definition and use of any parallel variable, for any possible

**Algorithm 1** The SFJ-based algorithm

```
procedure IDENTIFYFORKPOINTS(rootScope)
    forkPoints ← empty
    for all sentence ∈ TRAVERSEDEPTHFIRST(rootScope) do
        varName ← GETPARALLELVAR(sentence, rootScope)
        if varName ≠ empty then
            ADDELEMENT(forkPoints, sentence)
        end if
    end for
    return forkPoints
end procedure
procedure IDENTIFYJOINPOINTS(rootScope, forkPoints)
    joinPoints ← empty
    for all sentence ∈ forkPoints do
        varName ← GETPARALLELVAR(sentence)
        currSentence ← sentence
        scope ← true
        repeat
            useSentence ← GETFIRSTUSE(varName, currSentence)
            if useSentence ≠ empty then
                useScope ← GETSCOPE(useSentence)
                varScope ← GETSCOPE(sentence)
                if CHECKINCLUDED(useScope, varScope) then
                    ADDELEMENT(joinPoints, useSentence)
                    currSentence ← useSentence
                end if
            else
                scope ← false
            end if
        until scope ≠ true
    end for
    return joinPoints
end procedure
```

execution path. This step involves reusing the primitives of the target parallel library plus inserting glue code to invoke (if defined) the user's optimizations. The former sub-step also adapts the parallel code to the application structure prescribed by the library (e.g. subclassing certain API classes, generating extra artifacts, etc.).

Targeting libraries supporting D&C (e.g. Satin) mostly requires source-to-source translation, because sequential methods calls are individually and directly forked in the output code via fork library API functions. For libraries relying on master-worker or bag-of-tasks execution models (e.g. Doug Lea's framework, GridGain), in which *hierarchical* relationships between parallel tasks are not present, EasyFJP "flats" the task structure of the sequential code. Fig. 4 shows part of the GridGain code generated by EasyFJP from the BinSearch application shown in Fig. 3.

**Table 2.** SFJ-based algorithm: Helper functions

| **Signature** | **Functionality** |
|---|---|
| getParallelVar (aSentence,rootScope) | If *aSentence* assigns a recursive call to a parallel variable, the variable name 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. |
| checkIncluded (aScope,anotherScope) | Checks whether *aScope* is the same scope as *anotherScope* or is a descendant of it. |

GridGain materializes SFJ via Java futures. Lines 15-19 and line 21 represent fork and join points, respectively. Instances of BinSearchTask perform the subcomputations by calling BinSearchGridGain.search(int, int[], ExecutionContext) on individual pieces of the array. For simplicity, this code does not exploit the latest GridGain API since it is fairly more verbose than previous versions.

Finally, at **step 3**, programmers can non-invasively improve the efficiency of their parallel applications via *policies*, which are rules that throttle the amount of parallelism. This is the only manual step and, even when not measured yet, the effort to specify policies should be low as they capture common and *simple* optimizations. EasyFJP allows developers to specify policies based on the nature of both their applications (e.g. using thresholds/memoization) and the execution environment (e.g. avoiding many forks with large-valued parameters in a high-latency network). Policies are associated to fork points through external configuration and can be switched without altering parallelized codes. For instance, BinSearch could be made forking search provided array.length is above some threshold by implementing the shouldFork(ExecutionContext), otherwise the sequential version of search would be executed. ExecutionContext allows users to introspect execution at both the method level (e.g. accessing parameter values) and the application level (e.g. obtaining the current depth of the task tree). In other words, this object allows developers to access certain runtime information that refers to parallel aspects of the application under execution and use the information to specify tuning decisions.

## 3.1 Developing with EasyFJP: Considerations

Determining whether a user application will effectively benefit from using EasyFJP depends on a number of issues that developers should have in mind. First, feeding EasyFJP with a properly structured code does not necessarily ensures

```
class BinSearchGridGain{
 boolean searchSeq(int elem, int[] array){
  // Same as BinSearch.search(int, int[])
 }
 boolean search(int elem, int[] array){
  search(elem, array, initContext());
 }
 boolean search(int elem, int[] array, ExecutionContext ctx){
  if (!getPolicy(ctx.getMethod()).shouldFork(ctx))
   return searchSeq(elem, array);
  . . .
  Grid grid = GridFactory.getGrid();
  GridExecutorCallableTask exec = new GridExecutorCallableTask();
  int[] half1 = halfOne(array);
  GridTaskFuture<boolean> s1future =
    grid.execute(exec, new BinSearchTask(this, ctx, elem, half1));
  int[] half2 = halfTwo(array);
  GridTaskFuture<boolean> s2future =
    grid.execute(exec, new BinSearchTask(this, ctx, elem, half2));
  . . .
  return s1future.get() || s2future.get();
 }
}
```

**Fig. 4.** Example GridGain code automatically generated by EasyFJP

increased performance and applicability. The choice of parallelizing an application (or an individual method) depends on whether the method itself can exploit parallelism. In other words, the potential performance gains in parallelizing an application is subject to its computational requirements, which is a design factor that must be first addressed by the developer. EasyFJP automates the process of generating a parallel, tunable application "skeleton", but does not aim at automatically determining the portions of an application suitable for being parallelized. Furthermore, the choice of targeting a specific parallel backend is mostly subject to availability factors, i.e. whether an execution environment running the desired parallel library (e.g. GridGain) is available or not. For example, a novice developer would likely target a parallel library he knows is installed on a particular hardware, rather than the other way around.

Likewise, 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. Again, whether these patterns benefit a particular parallelized application depends on several factors. For example, not all FJP applications can exploit memoization techniques.

Moreover, an issue that may affect applicability is concerned with compatibility and interrelations with commonly-used techniques and libraries, such as multithreading and AOP. In a broad sense, these techniques literally alter the ordinary semantics of a sequential application. Particularly, multithreading makes deterministic sequential code non-deterministic, while AOP modifies the normal control flow of applications through the implicit use of artifacts containing aspect-specific behavior. Therefore, when using EasyFJP to parallelize such applications, various compatibility problems may arise depending on the backend selected for parallelization. Note that this is not an inherent limitation of EasyFJP, but of the target backend. Thus, before parallelizing an application with EasyFJP, a prior analysis should be carried out to determine whether the target parallel runtime is compatible with the libraries the application relies on.

## 4 EasyFJP implementation

The implementation of EasyFJP (`http://code.google.com/p/easyfjp-imp/`) is based on the notion of *Builder*. A Builder is a piece of code that concentrates knowledge on the use of a parallel library and therefore is responsible for the entire code generation process. The more the variety of Builders that are plugged into EasyFJP, the more the parallelization choices the tool offers to users who will use EasyFJP to write applications that take advantage of parallelism.

From a functional point of view, a Builder performs its work by relying on three basic components: a *code analyzer*, a target *parallel library* and a *code generator*. The *code analyzer* is the component in charge of identifying where to insert calls to the target *parallel library*. The output of this code analysis is the fork and join points. These points are required by the *code generator*, the component which performs the transformation of the original code into its parallelized counterpart by adding parallelization instructions into the target method. The parallelization instructions to support fork and join points are highly coupled to a *parallel library*, since the last one is the component that provides the parallelization support and acts during the actual execution of the application. The abstract design of a Builder was thought as a set of combinable and exchangeable components, to facilitate the extension of the tool. To goal is to enable EasyFJP to cover a wide range of parallel environments through the utilization of different parallel libraries that use different Fork-Join synchronization patterns and provide different code customizations to optimize parallel computations.

The parallelization process starts when the programmer indicates the Java class of his/her application, which contains the D&C method to be parallelized. Currently, this operation is done by writing a simple XML file. Then, the programmer needs to invoke a Java tool including a class called *Parallelizer* to start the automatic source code transformation, which comprises:

1. Peer Class Building: is the step in the parallelization process where fork and join points are identified and then converted into middleware API calls. The resulting artifact is the peer class.
2. Policy Injection: is the step where EasyFJP adds to the peer class the references to the policies optionally provided by programmers with experience in parallelization concepts.
3. Peer Class Binding: is the step through which the main application is bound to the peer class (i.e. the one built on step 1) so that every call to the sequential D&C method is forwarded to its parallelized counterpart.

It is worth clarifying the existing relation between the previously mentioned steps and Builder-related components. The *code analyzer*, which acts in the first step, will be described in detail below. The *code generator*, instead, is present each time the Java code is modified. Therefore, this component is needed not only to translate fork and join points into middleware API calls but also when extra logic in the shape of policies is planned to be added to the parallelized code, and finally, to establish the link between the sequential and the parallelized code of the application. Then, the component is used throughout the three steps. The classes that implement it will be described below. Lastly, the remaining component -the *parallel library*- plays a protagonic role in the first and second steps. However, despite being a component strongly related to the *code analyzer* and the *code generator*, the implementation is not part of EasyFJP. In other words, this is why EasyFJP rely on existing parallel libraries.

Fig. 5 shows the main classes of EasyFJP and the way they collaborate. The *Parallelizer* class is the entry point to the tool. It uses three collaborator classes to perform the steps described above. The Peer Class Building step is done by a set of classes that respond to the Gamma's Builder creational design pattern. It is composed by the *PeerClassDirector* class and the *PeerClassBuilder* interface. The former defines a generic algorithm to obtain the Peer Class as the final product. The algorithm uses the *PeerClassBuilder* interface to perform the steps it defines. These are mainly part of the *Code Analyzer* component, although some code, the one related to inserts middleware API calls, belong to the *Code Generator* component. Refining the previous algorithm by extending the *PeerClassDirector* class as well as providing an extension to the *PeerClassBuilder* interface, is the way to support SFJ and MFJ synchronization patterns. *SFJPeerClassDirector* and *SFJPeerClassBuilder* are examples of such extensions. In addition, the *code generator* component is also present in the *PolicyManager* and *BindingManager* classes. Both define generic procedures to achieve their purposes, i.e. the Policy Injection and the Peer Class Binding steps, respectively. These generic algorithms and procedures mentioned allows us to contemplate

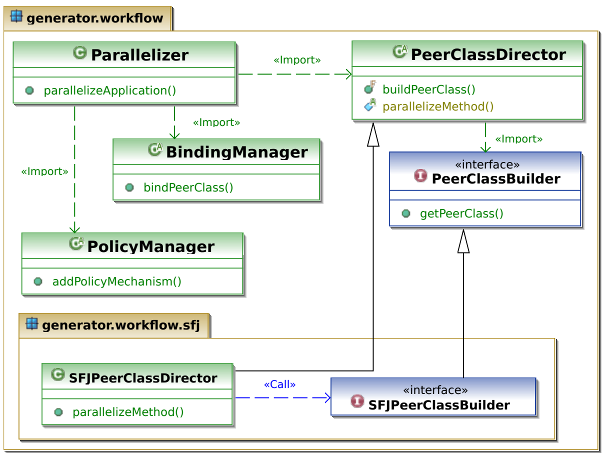


**Fig. 5.** EasyFJP main classes of the workflow package

the peculiarities of the target parallel library (i.e. execution environment initialization), and also the library used to manipulate the input Java code.

## 5 Experimental evaluation

The practical implications of using EasyFJP are determined by two main aspects, namely how competitive is implicitly supporting FJP synchronization patterns in D&C codes compared to explicit parallelism and classical parallel programming models, and whether policies are effective to tune parallelized applications or not. Hence, we conducted experiments in the context of the MFJ pattern in [17]. Furthermore, next we report experiments with SFJ through our new bindings to GridGain to further analyze the trade-offs behind using EasyFJP.

As a testbed, we used a three-cluster Grid emulated on a 15-machine LAN through WANem 2.2 with common WAN conditions. We tested a ray tracing and a gene sequence alignment application, whose parallel versions were obtained from sequential D&C codes from the Satin project. Apart from the challenging nature of the environment, the applications had high cyclomatic complexity, so they were representative to stress our code analysis mechanisms.

We fed the applications with various 3D scenes and real gene sequence databases from the National Center for Biotechnology Information (`http://www.ncbi.nlm.nih.gov`). For ray tracing, we used three task granularities: fine, medium and coarse, i.e. about 17, 2 and 1 parallel tasks per node, respectively. By "granularity" we refer to the amount of cooperative tasks in which a larger computation is split for execution. More tasks means finer granularities. Furthermore, for sequence alignment, we also employed three granularities, each with a number of tasks that depended on the size of the input database for efficiency purposes. For either application, we implemented two EasyFJP variants by using a threshold policy to regulate task granularity and another policy additionally exploiting data locality, a feature of EasyFJP to place tasks processing near parts of the input data in the same cluster. We developed hand-coded GridGain variants through its parallel annotations and its support for Google's MapReduce [18]. Fig. 6 and Fig. 7 illustrate the average running time (40 executions) of the ray tracing and the sequence alignment applications, respectively.

For ray tracing, the execution times uniformly increased as granularity became finer for all tests, which shows a good overall correlation of the different variants. For fine and medium granularities, EasyFJP was able to outperform their competitors since SFJ in conjunction with either policies achieved performance gains of up to 29%. For coarse granularities, however, the best EasyFJP variants introduced overheads of 1-9% with respect to the most efficient GridGain implementations. As expected, 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 becomes negligible for coarse-grained tasks. Again, the most efficient granularities were fine and medium in the sense they delivered the best data communication over processor usage ratio.

For sequence alignment, the running times were smaller as the granularity increased. Interestingly, like for ray tracing, EasyFJP obtained better performance for the fine granularity, and performed very competitively for the medium granularity. However, the GridGain variants were slightly more efficient when using coarse-grained tasks. In general, data locality did not help in reducing execution time because, unlike ray tracing, parallel tasks had a higher degree of independence. This does not imply that data locality policies are not effective but their usage should be decided depending on the nature of parallelized applications, which enforces similar previous findings [17].

## 6 Conclusions

EasyFJP 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 away from this latter. This separation, alongside with the simplicity of FJP and D&C, makes EasyFJP suitable for gradually mastering parallel programming.

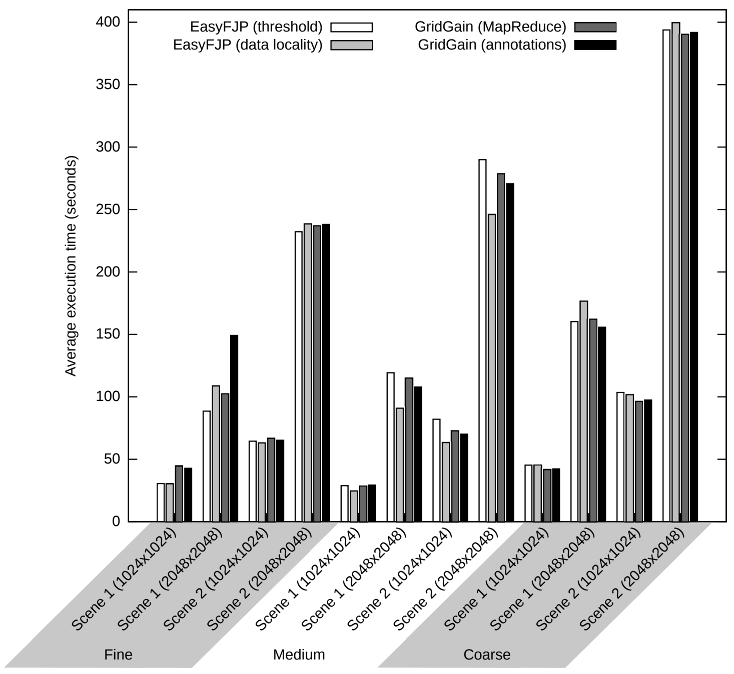


**Fig. 6.** Ray tracing: Average execution time

Our experimental results and the ones reported in [17] confirm that FJP-based implicit parallelism and policy-oriented explicit tuning, glued together via generative programming, are a viable approach to PaaC. We are however performing more experiments with more SFJ-based and MFJ-based parallel libraries to better ensure results validity, which is at present our main treat to validity. Moreover, EasyFJP has the potentiality to offer a better balance to the "ease of use and versatility versus performance" trade-off inherent to parallel programming tools for fine and medium-grained parallelism, plus the flexibility of generating code to exploit various parallel libraries. Up to now, EasyFJP deals with two broad parallel *concerns*, namely task synchronization and application tuning. We are adding other common parallel concerns such as inter-task communication, and adapting our ideas to newer parallel environments such as Clouds.

There is a recent trend that encourages researchers to create programming tools that simplify parallel software development by reducing the analysis and transformation burden when parallelizing sequential programs, which improves programmer productivity [19]. We are therefore building an IDE support to simplify the adoption and use of EasyFJP based on Eclipse. Finally, we have pro-

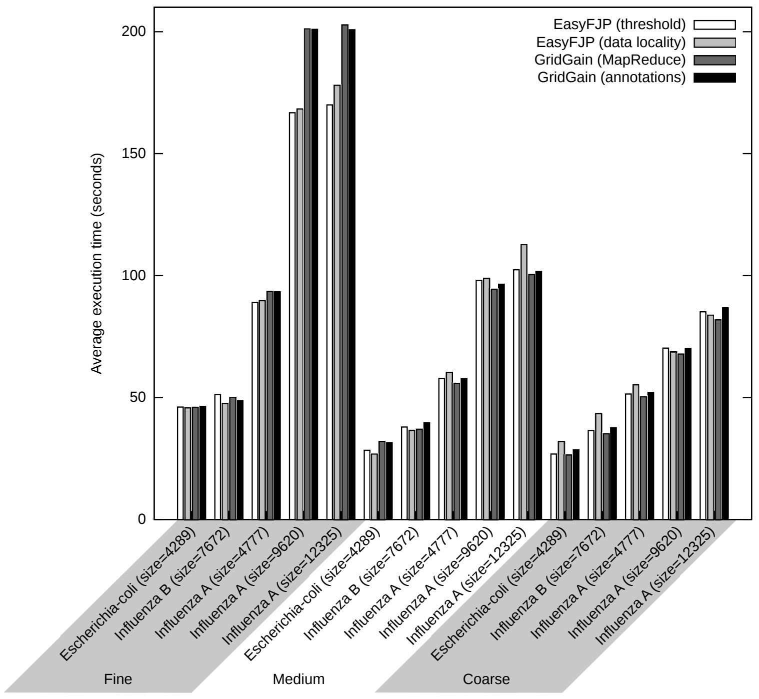


**Fig. 7.** Sequence alignment: Average execution time

duced a prototype to support the development of parallel applications within pure engineering communities, where scripting languages such as Python and Groovy are the common choice [20]. At present, we have redesigned the EasyFJP policy API and its associated runtime support to allows users to code policies in Python and Groovy [20], but this requires further research.

## References

1. Doug Lea. The java.util.concurrent synchronizer framework. *Science of Computer Programming*, 58(3):293–309, 2005.
2. John Danaher, I. Lee, and Charles Leiserson. Programming with exceptions in JCilk. *Science of Computer Programming*, 63(2):147–171, 2006.
3. Max Haustein and Klaus-Peter Lohr. JAC: Declarative Java concurrency. *Concurrency and Computation: Practice and Experience*, 18(5):519–546, 2006.
4. Rafael Duarte, Alexandre Mota, and Augusto Sampaio. Introducing concurrency in sequential Java via laws. *Information Processing Letters*, 111(3):129–134, 2011.
5. Muhammad Aleem, Radu Prodan, and Thomas Fahringer. JavaSymphony: A programming and execution environment for parallel and distributed many-core architectures. *Lecture Notes in Computer Science*, 6272:139–150, 2010.

6. Bao-Yin Zhang, Guang-Wen, Yang, and Wei-Min Zheng. Jcluster: An efficient Java parallel environment on a large-scale heterogeneous cluster. *Concurrency and Computation: Practice and Experience*, 18(12):1541–1557, 2006.
7. Hiu Chan, Andrew Gallagher, Appu Goundan, Yi Au Yeung, Aaron Keen, and Ronald Olsson. Generic operations and capabilities in the JR concurrent programming language. *Computer Languages, Systems and Structures*, 35(3):293–305, 2009.
8. Hua Zhang, Joohan Lee, and Ratan Guha. VCluster: A thread-based Java middleware for SMP and heterogeneous clusters with thread migration support. *Software: Practice and Experience*, 38(10):1049–1071, 2008.
9. Rob Van Nieuwpoort, Gosia Wrzesinska, Ceriel Jacobs, and Henri Bal. Satin: A high-level and efficient Grid programming model. *ACM Transactions on Programming Languages and Systems*, 32:9:1–9:39, 2010.
10. GridGain Systems. GridGain = Real Time Big Data. `http://www.gridgain.com`, 2011.
11. Sourceforge.net. Java Parallel Processing Framework. `http://www.jppf.org`, 2009.
12. Vincent Freeh. A comparison of implicit and explicit parallel programming. *Journal of Parallel and Distributed Computing*, 34(1):50–65, 1996.
13. Brian Amedro, Denis Caromel, Fabrice Huet, and V. Bodnartchouk. Java Proactive vs. Fortran MPI: Looking at the future of parallel Java. In *IPDPS'08*, pages 1–7. IEEE, 2008.
14. Cristian Mateos, Alejandro Zunino, and Marcelo Campo. On the evaluation of gridification effort and runtime aspects of JGRIM applications. *Future Generation Computer Systems*, 26(6):797–819, 2010.
15. Marco Aldinucci, Marco Danelutto, and Patrizio Dazzi. Muskel: An expandable skeleton environment. *Scalable Computing: Practice and Experience*, 8(4):325–341, 2007.
16. J. Sobral and A. Proença. Enabling JaSkel skeletons for clusters and computational Grids. In *CLUSTER'07*, pages 365–371. IEEE, 2007.
17. Cristian Mateos, Alejandro Zunino, and Marcelo Campo. An approach for non-intrusively adding malleable fork/join parallelism into ordinary JavaBean compliant applications. *Computer Languages, Systems and Structures*, 36(3):53–59, 2010.
18. Ralf Lämmel. Google's MapReduce programming model — revisited. *Science of Computer Programming*, 68(3):208–237, 2007.
19. Danny Dig. A refactoring approach to parallelism. *IEEE Software*, 28(1):17–22, 2011.
20. Cristian Mateos, Alejandro Zunino, Matías Hirsch, and Mariano Fernández. Enhancing the BYG gridification tool with state-of-the-art Grid scheduling mechanisms and explicit tuning support. *Advances in Engineering Software*, 43:27–43, 2012.