This article is a pre-print of the article "C. Mateos, A. Zunino and M. Hirsch: "Parallelism as a concern in Java through fork-join synchronization patterns". 4th TTSDP - ICCSA 2012. pp. 49-56. ISBN 978-0-7695-4710-7/12. IEEE. "
The published version is available at http://dx.doi.org/10.1109/ICCSA.2012.18

# Parallelism as a concern in Java through fork-join synchronization patterns

Cristian Mateos, Alejandro Zunino
*ISISTAN Research Institute - UNICEN University*
*Tandil (B7001BBO), Buenos Aires, Argentina*
*Tel.: +54 (2293) 439682.*
*Also CONICET*
*Email: cmateos@conicet.gov.ar*

Matías Hirsch
*UNICEN University*
*Tandil (B7001BBO), Buenos Aires, Argentina*

***Abstract*—We are facing a hardware revolution given by the increasing availability of multicore computers, clusters, Grids, and combinations of these. Consequently, there is plenty of computational power, but today's programmers are not fully prepared to exploit parallelism. Particularly, Java has helped in handling the heterogeneity of such environments, but there is a lack of 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). We briefly survey relevant Java-based parallel development tools, identify their drawbacks, and discuss some insights of an ongoing approach for mitigating them.**

***Keywords*-parallel software development; distributed computing; fork-join synchronization patterns; Java**

## I. Introduction

The advent of powerful parallel environments doubtlessly calls for new parallel programming tools. Many existing tools remain hard to use for an average programmer, and prioritize performance over other desirable attributes, i.e. code invasiveness and independence of the execution environment. 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 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 parallel tools have focused on running on *one* environment. Besides, they often offer developers APIs for programmatically coordinating parallel subcomputations. This requires knowledge on parallel (or distributed) programming, and output codes tied to the library used, compromising code maintainability and portability to other libraries.

This programmatic and intrusive approach to parallelism is also followed by several contemporary parallel languages such as Fortress (Oracle), X10 (IBM) and Axum (Microsoft), but it is still uncertain whether they will be widespread. In fact, other researchers propose extending conventional programming languages for parallelism instead of creating parallel languages from scratch, being Intel® TBB [1] and Apple's Grand Central Dispatch [2] examples of such dialects. 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.

## II. Parallelism in Java: background

To date, several Java tools for parallelizing CPU-hungry applications exist. Regarding multicore programming, Doug Lea's framework [3] and JCilk [4] extend the Java runtime library with concurrency primitives. Alternatively, JAC [5] separates application logic from thread management via annotations. Duarte et al. [6] 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 [7], JCluster [8], JR [9], VCluster [10] and Satin [11]). A distinctive feature of them compared to other Java libraries for building classical master-worker applications such as GridGain [12] or JPPF [13] 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.

This article is a pre-print of the article "C. Mateos, A. Zunino and M. Hirsch: "Parallelism as a concern in Java through fork-join synchronization patterns". 4th TTSDP - ICCSA 2012. pp. 49-56. ISBN 978-0-7695-4710-7/12. IEEE. "
The published version is available at http://dx.doi.org/10.1109/ICCSA.2012.18

Broadly, parallel programming is classified into implicit and explicit [14]. The former allows programmers to write applications without thinking about parallelism, which is automatically performed by the runtime system. However, 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 the concepts and the API of the parallel programming tool used. Besides, 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 (Figure 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 [5]), metaobjects (ProActive [15]) and Dependency Injection (JGRIM [16]). 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 [17], [18].

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 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 the concept of PaaC by adopting a base programming model providing opportunities for enabling *implicit* nevertheless versatile forms of parallelism, and by using generative programming to build code that leverages existing parallel libraries. Developers proficient in parallel programming can further optimize generated codes via an *explicit*, but non-invasive tuning framework.

## III. Fork-join parallelism to the rescue

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 [19]. In fact, Java, which has offered threads as first-class citizens for years, includes now an FJP framework for multicore CPUs (http://openjdk.java.net/projects/jdk7/features), which is based on Doug Lea's work.

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. More recently, Computational Grids, which arrange resources from geographically dispersed sites, have emerged as another environment for parallel computing. Interestingly, multicore CPUs, clusters and Grids alike can execute FJP tasks, as they conceptually comprise processing nodes (cores or individual machines) interconnected through communication "links" (a system bus, a high-speed LAN or a WAN). This uniformity arguably allows the same FJP application to be run in either environments, provided there is a platform aware of the underlying execution support.

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's result. With MFJ, the programmer waits for the results of the tasks launched up to

This article is a pre-print of the article "C. Mateos, A. Zunino and M. Hirsch: "Parallelism as a concern in Java through fork-join synchronization patterns". 4th TTSDP - ICCSA 2012. pp. 49-56. ISBN 978-0-7695-4710-7/12. IEEE. "
The published version is available at http://dx.doi.org/10.1109/ICCSA.2012.18

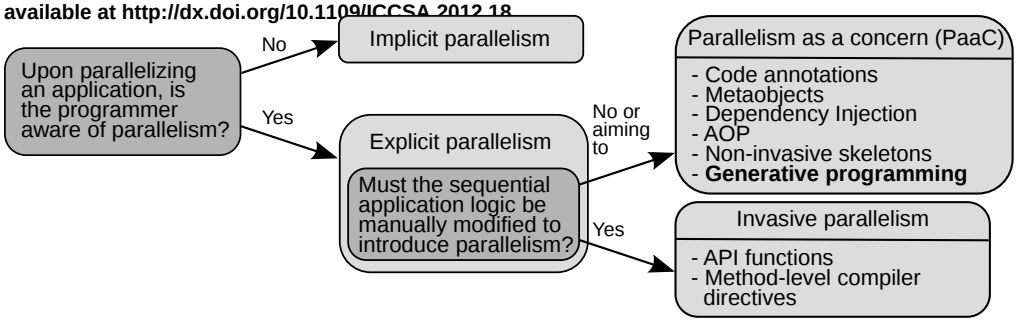


Figure 1. Parallelism in Java: Taxonomy

a synchronization call. In the following codes, two SFJ calls are necessary to safely access the results of $task_1$ and $task_2$ (top), whereas the same behavior is achieved with one MFJ call (bottom):

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

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

Examples of Java-based parallel libraries and their support for synchronization 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.

## IV. FJP as a concern: EasyFJP

Intuitively, FJP is suitable for parallelizing divide and conquer (D&C) applications, an algorithmic abstraction useful to solve many problems. Our ongoing EasyFJP project [20] is to design source code analysis algorithms and code generation techniques to automatically introduce SFJ and MFJ into sequential D&C applications. EasyFJP comprises a semi-automatic parallelization process (Figure 2) that outputs library-dependent parallel codes with hooks for attaching user optimizations.

### A. Step 1

First, the sequential application is analyzed to spot the points within the target method that perform recursive calls and access to recursive results. These dependencies are guarded to keep the correctness of the code. Before feeding EasyFJP, programmers only have to assign results to local variables. Then, depending on the target parallel library, EasyFJP uses an MFJ or a SJF-inspired algorithm to detect prospective fork and join points. Either parallelization algorithms behave heuristically, for example by minimizing the derived join points. For brevity, below we discuss the SFJ algorithm; [20] presents a preliminary version of its MFJ counterpart.

The SJF-based algorithm (see Algorithm 1 and Table I) 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. Based on the identified join points, EasyFJP modifies the source code to call a library-specific join primitive between the definition and use of any parallel variable, for any possible execution path. As input, the algorithm operates on a tree derived from the target method's code. Nodes in this tree are method scopes, while arcs represent relationships between scopes.

### B. Step 2

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. extends from certain API classes, generates extra artifacts, etc.).

This article is a pre-print of the article "C. Mateos, A. Zunino and M. Hirsch: "Parallelism as a concern in Java through fork-join synchronization patterns". 4th TTSDP - ICCSA 2012. pp. 49-56. ISBN 978-0-7695-4710-7/12. IEEE. "
The published version is available at http://dx.doi.org/10.1109/ICCSA.2012.18

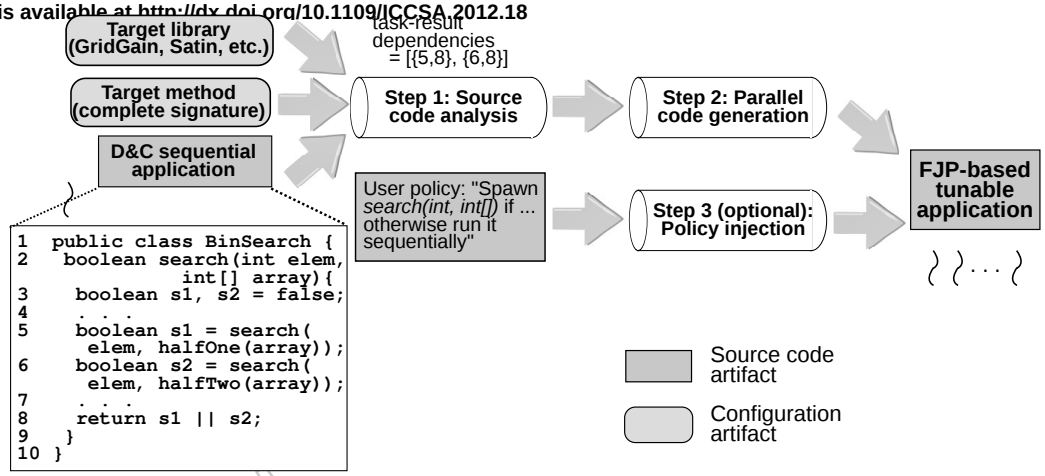


Figure 2. EasyFJP: Parallelization process

Table I
SFJ-BASED ALGORITHM: HELPER FUNCTIONS

| Signature | Functionality |
|---|---|
| getParallelVar(aSentence,rootScope) | Checks whether *aSentence* assigns a recursive call to a parallel variable. If so, the name of the variable 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. |

Targeting libraries supporting D&C such as Satin mostly requires source-to-source translation, as sequential methods are forked in the output code via proper library API calls. For libraries relying on master-worker or bag-of-tasks execution models (e.g. Doug Lea's framework and GridGain), in which there are not *hierarchical* relationships between parallel tasks, EasyFJP "flats" the task structure of the sequential application. For illustrative purposes, Figure 3 shows part of the GridGain code generated by EasyFJP[1] from the BinSearch application shown in Figure 2.

GridGain materializes SFJ through Java futures. Lines 19-22 and line 24 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 the sake of simplicity, this code does not exploit the latest version of the GridGain API (i.e. 3.5.0) since it is fairly more verbose than its previous versions.

[1]The current version of the GridGain code generator of EasyFJP is available at http://code.google.com/p/easyfjp-imp/

### C. Step 3

Programmers can non-invasively improve the efficiency of their parallel applications via *policies*, which are rules that throttle the amount of parallelism. 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, otherwise the sequential version of search is executed:

```
import easyfjp.core.Policy;
import easyfjp.core.ExecutionContext;

class MyThresholdPolicy implements Policy{
 static int MIN_ARRAY_SIZE = 50;
 boolean shouldFork(ExecutionContext ctx){
  int[] firstArgument = (int[])ctx.getArgument(1);
  return firstArgument.length > MIN_ARRAY_SIZE;
```

This article is a pre-print of the article "C. Mateos, A. Zunino and M. Hirsch: "Parallelism as a concern in Java through fork-join synchronization patterns". 4th TTSDP - ICCSA 2012. pp. 49-56. ISBN 978-0-7695-4710-7/12. IEEE. "
The published version is available at http://dx.doi.org/10.1109/ICCSA.2012.18

```
1  import org.gridgain.grid.Grid;
2  import org.gridgain.grid.GridFactory;
3  import org.gridgain.grid.GridTaskFuture;
4  import org.gridgain.grid.kernal.executor.GridExecutorCallableTask;
5
6  class BinSearchGridGain{
7   boolean searchSeq(int elem, int[] array){
8    // Same as BinSearch.search(int, int[])
9   }
10  boolean search(int elem, int[] array){
11   search(elem, array, initContext());
12  }
13  boolean search(int elem, int[] array, ExecutionContext ctx){
14   if (!getPolicy(ctx.getMethod()).shouldFork(ctx))
15    return searchSeq(elem, array);
16   . . .
17   Grid grid = GridFactory.getGrid();
18   GridExecutorCallableTask exec = new GridExecutorCallableTask();
19   GridTaskFuture<boolean> s1future =
20     grid.execute(exec, new BinSearchTask(this, ctx, elem, getFirstHalf(array)));
21   GridTaskFuture<boolean> s2future =
22     grid.execute(exec, new BinSearchTask(this, ctx, elem, getSecondHalf(array)));
23   . . .
24   return s1future.get() || s2future.get();
25  }
26 }
```

Figure 3. Example GridGain code automatically generated by EasyFJP

```
 }
}
```

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.

## V. Developing with EasyFJP: Further issues

Up to this point, we have described the parallelization and tuning process promoted by EasyFJP to parallelize sequential codes. However, determining whether a user application will effectively benefit from using EasyFJP depends on a number of issues that developers should have in mind. This section explains what users should expect from our approach and what not.

### A. Performance and backend selection

Feeding EasyFJP with a properly structured code does not necessarily mean it will benefit from our parallelization process or even the process will be applicable. 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 its nature. For example, just a subset of FJP applications can exploit memoization techniques.

### B. Applicability

An issue that may affect applicability is concerned with compatibility and interrelations with commonly-

This article is a pre-print of the article "C. Mateos, A. Zunino and M. Hirsch: "Parallelism as a concern in Java through fork-join synchronization patterns". 4th TTSDP - ICCSA 2012. pp. 49-56. ISBN 978-0-7695-4710-7/12. IEEE. "
The published version is available at http://dx.doi.org/10.1109/ICCSA.2012.18

**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
```

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 [19], 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.

### C. Debuggability

Using EasyFJP does not differ from the pack in terms of debuggability, in which parallel programming has been historically conceived as a notoriously hard task. When not using policies, debugging EasyFJP applications that target certain backends should be as difficult as debugging the counterparts obtained by manually using those backends. Policies may make debugging more complex as they change the operational semantics of a program. Nevertheless, this problem is also shared by approaches to parallel development based on separating the functional behavior of a program from its parallel concerns, such as those tools that rely on AOP techniques or rules to parallelize/tune applications. Interestingly, both these approaches and EasyFJP arguably ease the task of testing the algorithmic correctness of programs prior to parallelization, which is more difficult to achieve with intrusive parallelization tools.

## VI. EVALUATION

From a pragmatic perspective, 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 [20]. Complementary, next we report experiments with SFJ through our 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 version 2.2 (http://wanem.sourceforge.net) 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 a high cyclomatic complexity, thus 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 parallel 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-

This article is a pre-print of the article "C. Mateos, A. Zunino and M. Hirsch: "Parallelism as a concern in Java through fork-join synchronization patterns". 4th TTSDP - ICCSA 2012. pp. 49-56. ISBN 978-0-7695-4710-7/12. IEEE. "
The published version is available at http://dx.doi.org/10.1109/ICCSA.2012.18

coded GridGain variants through its parallel annotations and its support for Google's MapReduce [21]. Figures 4 and 5 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 [20].

## VII. 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 enter the world of parallel programming.

Our experimental results and the ones reported in [20] confirm that FJP-based implicit parallelism and policy-oriented explicit tuning, glued together via generative programming, are a viable approach to PaaC. 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. This will certainly more in-depth future research.

Moreover, there is a recent trend that encourages researchers to create 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 [22]. In this line, we are building an 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 scripting languages such as Python and Groovy are the common choice [23]. 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 all EasyFJP concepts directly into these scripting languages. Then, we will investigate how to port and exploit the parallelization heuristics of EasyFJP apart from its policy mechanism.

## References

[1] W. Kim and M. Voss, "Multicore desktop programming with Intel Threading Building Blocks," *IEEE Software*, vol. 28, no. 1, pp. 23–31, Jan. 2011.

[2] V. Nahavandipoor, *Concurrent Programming in Mac OS X and iOS: Unleash Multicore Performance with Grand Central Dispatch*. O'Reilly Media, May 2011.

[3] D. Lea, "The java.util.concurrent synchronizer framework," *Science of Computer Programming*, vol. 58, no. 3, pp. 293–309, 2005.

[4] J. Danaher, I. Lee, and C. Leiserson, "Programming with exceptions in JCilk," *Science of Computer Programming*, vol. 63, no. 2, pp. 147–171, 2006.

This article is a pre-print of the article "C. Mateos, A. Zunino and M. Hirsch: "Parallelism as a concern in Java through fork-join synchronization patterns". 4th TTSDP - ICCSA 2012. pp. 49-56. ISBN 978-0-7695-4710-7/12. IEEE. "
The published version is available at http://dx.doi.org/10.1109/ICCSA.2012.18

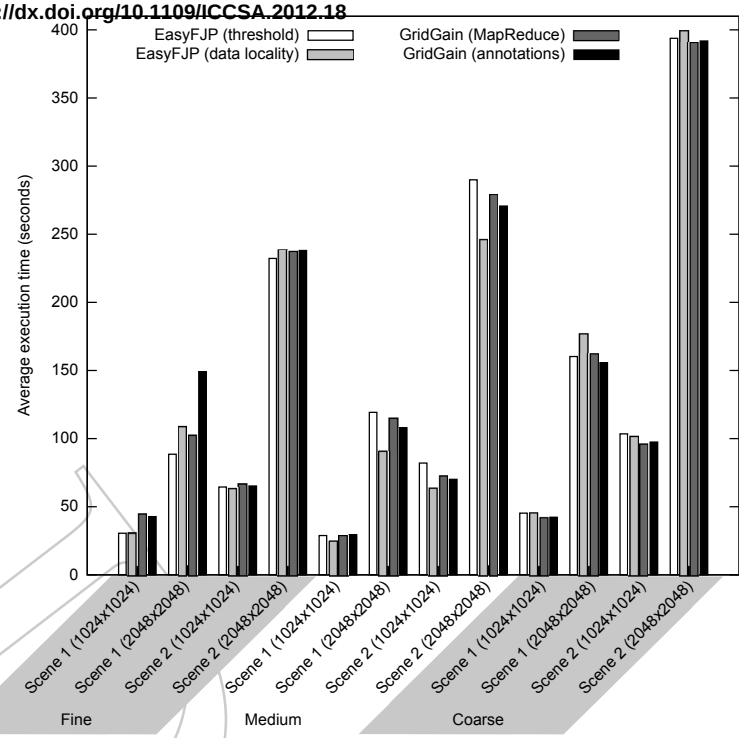


Figure 4. Ray tracing: Average execution time

[5] M. Haustein and K.-P. Lohr, "JAC: Declarative Java concurrency," *Concurrency and Computation: Practice and Experience*, vol. 18, no. 5, pp. 519–546, 2006.

[6] R. Duarte, A. Mota, and A. Sampaio, "Introducing concurrency in sequential Java via laws," *Information Processing Letters*, vol. 111, no. 3, pp. 129–134, 2011.

[7] M. Aleem, R. Prodan, and T. Fahringer, "JavaSymphony: A programming and execution environment for parallel and distributed many-core architectures," in *Euro-Par 2010 - Parallel Processing*, ser. Lecture Notes in Computer Science. Springer, 2010, vol. 6272, pp. 139–150.

[8] B.-Y. Zhang, Z.-Y. Mo, G.-W. Yang, and W.-M. Zheng, "Dynamic load-balancing and high performance communication in JCluster," in *International Parallel and Distributed Processing Symposium*. IEEE, 2007, p. 227.

[9] H. Chan, A. Gallagher, A. Goundan, Y. Au Yeung, A. Keen, and R. Olsson, "Generic operations and capabilities in the JR concurrent programming language," *Computer Languages, Systems and Structures*, vol. 35, no. 3, pp. 293–305, 2009.

[10] H. Zhang, J. Lee, and R. Guha, "VCluster: A thread-based Java middleware for SMP and heterogeneous clusters with thread migration support," *Software: Practice and Experience*, vol. 38, no. 10, pp. 1049–1071, 2008.

[11] R. Van Nieuwpoort, G. Wrzesinska, C. Jacobs, and H. Bal, "Satin: A high-level and efficient Grid programming model," *ACM Transactions on Programming Languages and Systems*, vol. 32, pp. 9:1–9:39, 2010.

[12] GridGain Systems, "GridGain = Real Time Big Data," http://www.gridgain.com, 2011.

[13] Sourceforge.net, "Java Parallel Processing Framework," http://www.jppf.org, 2009.

[14] V. Freeh, "A comparison of implicit and explicit parallel programming," *Journal of Parallel and Distributed Computing*, vol. 34, no. 1, pp. 50–65, 1996.

[15] B. Amedro, D. Caromel, F. Huet, and V. Bodnartchouk, "Java Proactive vs. Fortran MPI: Looking at the future of parallel Java," in *International Parallel and Distributed Processing Symposium*. IEEE, 2008, pp. 1–7.

[16] C. Mateos, A. Zunino, and M. Campo, "On the evaluation of gridification effort and runtime aspects of JGRIM applications," *Future Generation Computer Systems*, vol. 26, no. 6, pp. 797–819, 2010.

[17] M. Aldinucci, M. Danelutto, and P. Dazzi, "Muskel: An expandable skeleton environment," *Scalable*

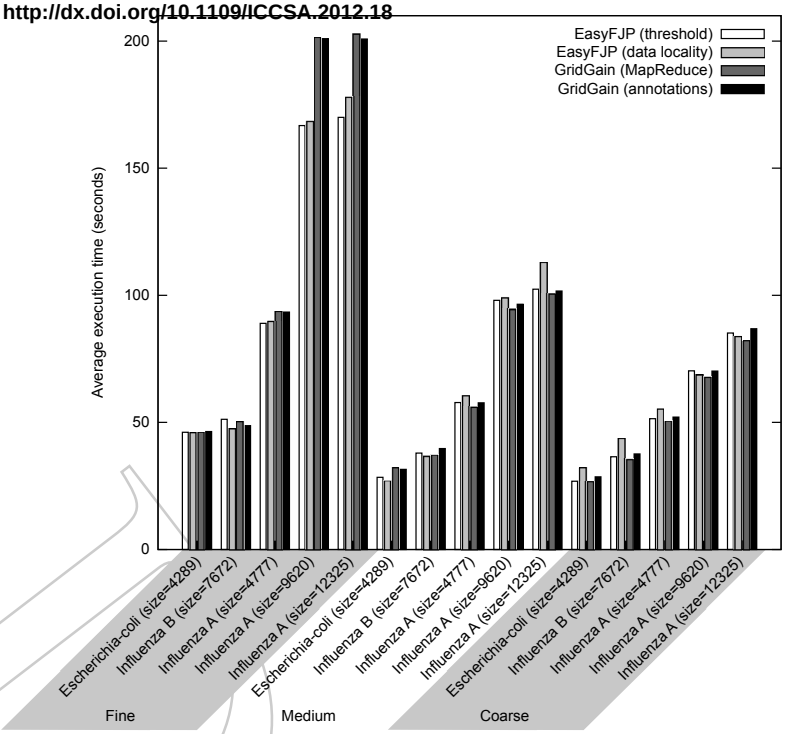


Figure 5. Sequence alignment: Average execution time

*Computing: Practice and Experience*, vol. 8, no. 4, pp. 325–341, 2007.

[18] J. Sobral and A. Proença, "Enabling JaSkel skeletons for clusters and computational Grids," in *International Conference on Cluster Computing*. IEEE, 2007, pp. 365–371.

[19] E. Lee, "The problem with threads," *Computer*, vol. 39, no. 5, pp. 33–42, 2006.

[20] C. Mateos, A. Zunino, and M. Campo, "An approach for non-intrusively adding malleable fork/join parallelism into ordinary JavaBean compliant applications," *Computer Languages, Systems and Structures*, vol. 36, no. 3, pp. 53–59, 2010.

[21] R. Lämmel, "Google's MapReduce programming model — revisited," *Science of Computer Programming*, vol. 68, no. 3, pp. 208–237, 2007.

[22] D. Dig, "A refactoring approach to parallelism," *IEEE Software*, vol. 28, no. 1, pp. 17–22, 2011.

[23] C. Mateos, A. Zunino, M. Hirsch, and M. Fernández, "Enhancing the BYG gridification tool with state-of-the-art Grid scheduling mechanisms and explicit tuning support," *Advances in Engineering Software*, vol. 43, pp. 27–43, Jan. 2012.