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

Matías Hirsch Alejandro Zunino Cristian Mateos

ISISTAN Research Institute.
Universidad Nacional del Centro de la Provincia de Buenos Aires (UNICEN)
Campus Universitario, Tandil (B7001BBO)
Buenos Aires, Argentina
Also CONICET (Consejo Nacional de Investigaciones Científicas y Técnicas)
e-mail: matias.hirsch@isistan.unicen.edu.ar
Tel. +54-2293-440363, ext. 35

Abstract

Because of the increasing availability of multi-core machines, clusters, Grids, and combinations of these environments, there is now plenty of computational power available for executing compute intensive applications. However, because of the overwhelming and rapid advances in distributed and parallel hardware and environments, 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 and tools to easily distributing and parallelizing applications. One solution to mitigate this problem and make some progress towards producing general tools seems to be the synthesis of semi-automatic parallelism and Parallelism as a Concern (PaaC), which allows 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, which precisely exploit these new concepts.

Keywords: Parallel software development, distributed and parallel computing, PaaC, fork-join synchronization patterns, Java, EasyFJP

1 Introduction and problem statement

The existence of compute intensive applications present in a wide range of domains including the entertainment industry, meteorology, economy, biology, physics, among others, and 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 are based on the traditional conception that high performance is the utmost goal, ignoring 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 parallel programming world. Low code invasiveness and environment neutrality are also important since they allow for hiding parallelism and distribution from the pure application logic of these domain-specific applications.

In dealing with the software diversity of such environments –specially distributed ones– Java is very interesting as it offers platform independence and competitive performance compared to conventional languages (Shafi, Carpenter, Baker, & Hussain, 2009) (Taboada, Ramos, Expósito, Touriño, & Doallo, 2011). However, most Java tools have focused on running on one environment exclusively, i.e., one of multi-core machines, clusters or Grids. Besides, they often offer developers APIs for programmatically coordinating subcomputations, but not parallel code generation techniques. This needs knowledge on parallel/distributed programming, and output codes are tied to the API 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 –i.e., applicable to several domains– delivering minimum development effort and code intrusiveness.

To date, several Java tools for scaling out CPU-hungry applications have been proposed in the literature. Regarding multi-core programming, Doug Lea's framework (Lea, 2005) and JCilk (Danaher, Lee, & Leiserson, 2006) extend the Java runtime library with concurrency primitives. Alternatively, JAC (Haustein & Lohr, 2006) aims at separating application logic from thread declaration and synchronization via regular Java annotations, with

a special emphasis on removing the differences between sequential and concurrent codes. Furthermore, Duarte et al. (Duarte, Mota, & Sampaio, 2011) address the same goal by automatically deriving thread-enabled source codes from sequential ones based on algebraic laws. Similarly, JOMP (Bull & Kambites, 2000) is compliant to OpenMP (Chandra, Dagum, Kohr, Maydan, McDonald, & Menon, 2000), a set of standard method-level/sentence-level directives and library routines for shared memory parallel programming, which is very popular.

Regarding cluster and Grid programming, most of the tools offer APIs to manually create and coordinate parallel computations. Some representative examples of such tools are JavaSymphony (Aleem, Prodan, & Fahringer, 2010), a 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 within their parallelized codes, JCluster (Zhang, Guang-Wen, Yang, & Zheng, 2006) -which supports the execution of task-oriented parallel applications in heterogeneous clusters. Tasks are scheduled according to the novel transitive random stealing algorithm, JR (Chan, Gallagher, Goundan, Au Yeung, Keen, & Olsson, 2009), which provides a rich concurrency model supporting remote JVM and object creation, asynchronous communication and rendezvous, and VCluster (Zhang, Lee, & Guha, 2008), a library that executes thread-based applications on clusters. In VCluster, threads migrate between nodes for load balancing purposes. Inter-thread communication is performed through virtual channels, which isolate threads location. Finally, Satin (Van Nieuwpoort, Wrzesinska, Jacobs, & Bal, 2010) is a library for parallelizing divide and conquer codes on LANs and WANs that follows the semantics of JCilk. A distinctive feature of these tools compared to other Java libraries for building classical master-worker applications such as GridGain (Systems, 2011) or JPPF (Sourceforge.net, 2009) is that the former support complex parallel applications structures in terms of code design. All in all, tools in both groups are designed for programming parallel codes rather than semi-automatically or automatically transforming sequential codes to cluster and Grid-aware ones.

Irrespective of the target execution environment, according to a well-known taxonomy in the area, parallel programming can be classified into implicit and explicit (Freeh, 1996). The former methodology allows programmers to write applications without thinking about parallelism and leaving parallel technical details on the background, which are dealt with automatically by the runtime system. However, performance of implicit parallelism may be suboptimal since programmers have no control over parallel subcomputations directly. Explicit parallelism on the other hand supplies APIs so that developers have more control over parallel execution to implement efficient applications, but the burden of managing parallelism falls on them, which involves more programming and testing costs. From the work analyzed in this Section, it follows that although they are designed with simplicity in mind, most of them are still inspired by explicit parallelism. Parallelizing applications then 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, i.e., exploiting certain characteristics of the underlying computational resources. This makes such tuning logic obsolete when an application is ported for example from a cluster to a Grid, since execution conditions are inherently different.

An alternative approach to traditional explicit parallelism is to treat parallelism as a *concern* (as in aspect oriented programming - AOP), thus avoiding mixing application logic with code implementing parallel behavior. As Table 1. Parallelism in Java: Taxonomy. Adapted from (Mateos, Zunino, & Campo, 2010) shows, this has gained momentum as reflected by Java tools that partly or entirely rely on mechanisms for separation of concerns, e.g., code annotations in JAC (Haustein & Lohr, 2006), metaobjects in ProActive (Amedro, Caromel, Huet, & Bodnartchouk, 2008), and Dependency Injection in JGRIM (Mateos, Zunino, & Campo, 2010b) (Mateos, Zunino, & Campo, 2008). 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. Approaches to instantiate these skeletons include wrapping sequential codes, or specializing framework classes as in (Aldinucci, Danelutto, & Dazzi, 2007) (Sobral & Proença, 2007).Error: No se encuentra la fuente de referencia

	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 - Metaobjects - Dependency Injection - AOP - Non-invasive Skeletons - **Generative programming**	- API functions - Method-level compiler directives

Table 1. Parallelism in Java: Taxonomy. Adapted from (Mateos, Zunino, & Campo, 2010).

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 multi-core machines. Moreover, approaches based on annotations require explicit modifications to insert parallelism and application-specific optimizations that obscure final codes. Metaobjects and specially AOP have helped in coping with this problem, but at the expense of incepting another programming paradigm that has to be learnt by programmers prior to parallelization. Lastly, tools providing support for various parallel patterns offer 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 compute-intensive applications through which the difficult and intrusive nature of parallelism is mitigated. 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 (Mateos, Zunino, & Campo, 2010). 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, the set of parallel primitives that represents the cornerstones of our approach. After that, Section 3 overviews the EasyFJP project and its main technical aspects regarding the materialization of the approach. Then, in Section 4 an implementation of EasyFJP is explained in detail. An empirical validation of EasyFJP implementation with several variants is reported in Section 5. Finally, Section 6 presents some concluding remarks.

2 An overview of Fork-join parallelism

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. Conceptually, FJP represents an alternative to threads, which have received criticism due to their inherent complexity in terms of program testing effort. In fact, Java, which has offered threads as first-class citizens for years, includes from version 7 an FJP framework for exploiting multi-core CPUs, which is essentially based on the well-known Doug Lea's framework.

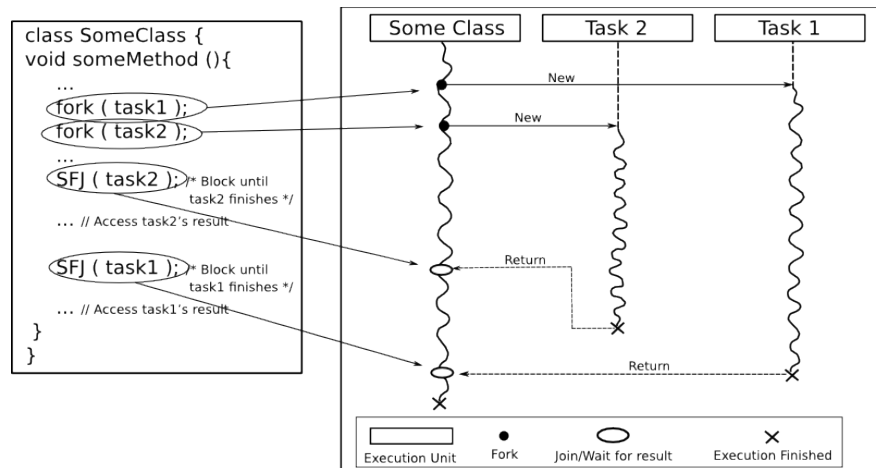

Fig. 1 Simple Fork-Join synchronization pattern

Interestingly, FJP is not circumscribed to multi-core programming, but is also applicable to any parallel or distributed execution environments where the notions of "tasks" and "processors" exist. For instance, forked tasks can be run on the machines of a cluster. Moreover, 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 executed in either environment by using environment-specific execution platforms to process the associated forked tasks. Broadly, current Java parallel libraries relying on task-oriented execution models offer API primitives to create one parallel task or a list of tasks simultaneously, which are firstly mapped to library-level execution units. As a complement to these primitives, these parallel libraries also expose primitives or use models to synchronize the access to the results of finalized tasks. Hence, the mechanism for parallelizing based on primitives to create tasks and coordinating results could be directly mapped to a Fork-Join pattern where Fork is the way to express parallelism and Join the way to access to the results. There are, however, operational differences among libraries concerning the primitives to synchronize sub-computations. 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. An example of this type of syncronization is the *Future* object, whose class is included into the *java.util.concurrent library*. These objects are used by GridGain and represent the result of an asynchronous computation. With MFJ, the programmer waits for the results of the tasks launched up to a synchronization call. The *Sync* primitive from Satin project is an example of MJF synchronization points. To better illustrate the idea, in the following codes, two SFJ calls are necessary to safely access the results of $task_1$ and $task_2$ (

), whereas the same behavior is achieved with one MFJ call (Fig. 2).

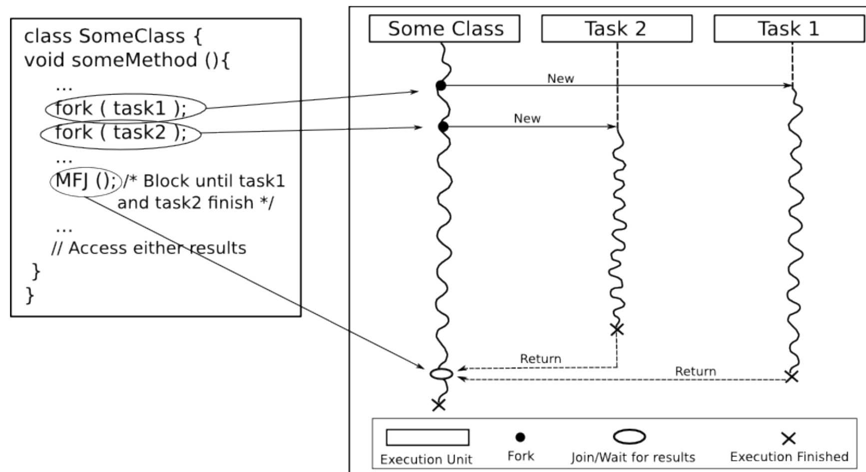

Fig. 2 Multiple Fork-Join synchronization pattern

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. In general, at least in Java, SFJ is more popular and it is implemented by most parallel libraries. However, as discussed, using libraries requires learning 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 sequential recursive invocations and the use of recursive results respectively. The EasyFJP project (Mateos, Zunino, & Campo, 2010) goals is precisely 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. Moreover, for the D&C version of the Binary Search code shown in Fig. 3 that serves as input of the EasyFJP parallelization process, there are two recursive calls or Fork points (lines 5 and 6) and two accesses to recursive results or Join points (line 8).

Broadly speaking, 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 recursive calls to local variables. Depending on the target parallel library selected, EasyFJP uses an MFJ or a SFJ-inspired algorithm to detect fork and join points, but the algorithms themselves do not depend on the parallel library selected. For brevity, below we discuss the SFJ algorithm, while (Mateos, Zunino, & Campo, 2010) presents its MFJ counterpart. As such, the fork-join pattern supported by this algorithm represents the main difference between this work and (Mateos, Zunino, & Campo, 2010).

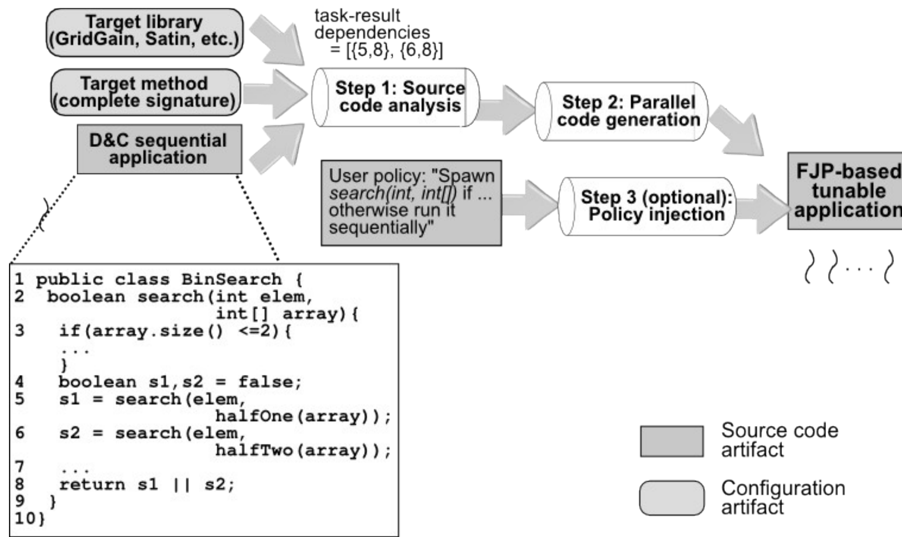

Fig. 3 EasyFJP: Parallelization process

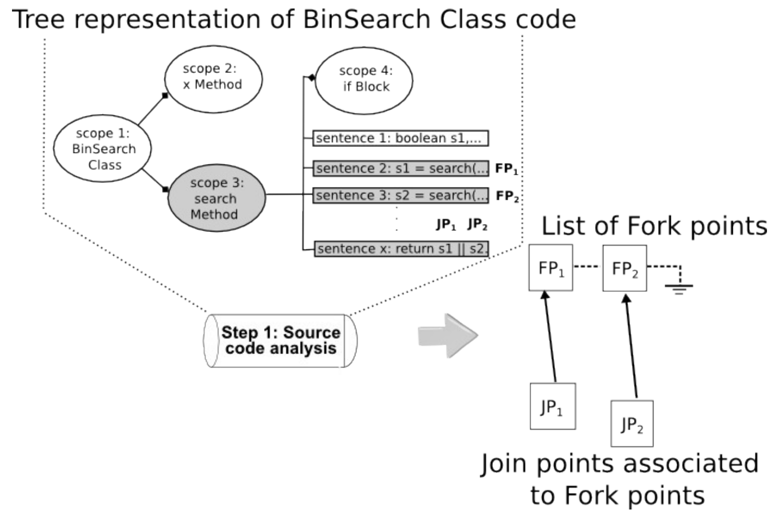

Fig. 4 Output sample of step 1

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 in a statement. As input, the algorithm operates on a tree derived from the target method source code (Fig. 4). Nodes in this tree are method scopes, while ancestor-descendant relationships represent nested scopes. First, the procedure *IdentifyForkPoints* search for parallel variables, that are local variables placed on the left side of an assignment operation where the right side is a recursive call. These recursive calls are the fork points. Once the list of fork points is identified, the associated list of join points has to be built. This is done by the *IdentifyJoinPoints* procedure, which is invoked with the list of fork points as argument. Thus, for every fork point, the algorithm performs an examination of the sentences looking for every use of the result of the parallel variable associated to a fork point. All the resulting occurrences are marked as join points of the current fork point under analysis. Finally, the algorithm passes on to step 2 the list of recursive calls and its corresponding uses of recursive results (fork and join points, Fig. 4) so that it can map them into parallel 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 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 (step 3). 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., GridGain or JPPF), in which *hierarchical* relationships between parallel tasks are not present, EasyFJP somewhat "flats" the task structure of the sequential source code.

shows part of the GridGain code generated by EasyFJP from the BinSearch application shown in Fig. 3.

GridGain materializes SFJ via Java futures. Lines 15-17 represent fork points while in line 19 join points have been translated into appropriate GridGain library API calls. Instances of BinSearchTask perform the subcomputations by calling BinSearchGridGain.search(int, int[], ExecutionContext) on individual pieces of the input array. For the sake of simplicity, this parallel code does not exploit the latest GridGain API since it is fairly more verbose than previous versions.

The SJF-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 joinPoints
    end procedure


    procedure IdentifyJoinPoints(rootScope, forkPoints)
        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(joinPoints,varScope) then
                        addElement(joinPoints, useSentence)
                        currSentence ← useSentence
                    end if
                else
                    scope ← false
                end if
            until scope ≠ true
        end for
        return joinPoints
end procedure
```

Alg. 1 The SJF-based algorithm

Finally, at step 3, programmers can optionally and non-invasively improve the efficiency of their parallel applications via *policies*, which are rules that regulate the amount of parallelism, or in other words the number of parallel tasks executing in the environment to handle the whole application. This is the only manual step and, even when not measured yet, the effort to specify *policies* is intuitively low as they capture common and *simple* optimizations so far.

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.

Table 2 SF-based fork and join points detection: Helper functions

```
1 class BinSearchGridGain{
2  boolean searchSeq(int elem, int[] array){
3   // Same as BinSearch.search(int, int[])
4  }
5  boolean search(int elem, int[] array){
6    search(elem, array, initContext());
7  }
8  boolean search(int elem, int[] array, ExecutionContext ctx){
9    if (!getPolicy(ctx.getMethod()).shouldFork(ctx))
10      return searchSeq(elem, array);
11     . . .
12     Grid grid = GridFactory.getGrid();
13     GridExecutorCallableTask exec = new GridExecutorCallableTask();
14     int[] half1 = halfOne(array);
15     GridTaskFuture<boolean> s1future = grid.execute(exec, new
   BinSearchTask(this, ctx, elem, half1));
16     int[] half2 = halfTwo(array);
17     GridTaskFuture<boolean> s2future = grid.execute(exec, new
   BinSearchTask(this, ctx, elem, half2));
18     . . .
19     return s1future.get() || s2future.get();
20  }
21}
```

Fig. 5 Example of GridGain code automatically generated by EasyFJP

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 files and can be switched without altering parallelized codes. For instance, BinSearch could be made forking search provided array.length is above an appropriate threshold by implementing the shouldFork(ExecutionContext), otherwise the sequential version of the method would be executed. This prevents using parallelism for small-sized arrays and falling back to

sequential execution to ensure good performance. ExecutionContext allows users to introspect execution at both the method level, such as accessing parameter values, and the application level, for example obtaining the current depth of the task hierarchy 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. Fig. 6 shows a possible implementation of a Threshold policy that, based on the input array size, which is part of the application context, decides whether or not to continue parallelizing the execution of the target method. Furthermore,
line 9 shows the glue code to illustrate how the parallelized BinSearch code references to a user-defined threshold policy.

```
1 public class ThresholdPolicy implements Policy{
2
3  final int MIN_LENGHT = 1000;
4
5  public boolean shouldFork(ExecutionContext contenxt){
6    int[] array = (int[]) context.getApplicationContext();
7    return array.lenght > MIN_LENGHT;
8  }
9 }
```

Fig. 6 Example of a threshold policy code

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 D&C code does not necessarily ensures increased performance and applicability. The choice of parallelizing an application (or an individual method) depends on whether the method itself can inherently exploit parallelism. In other words, the potential performance gains after parallelizing an application is subject to its computational requirements, which is a design factor that must be first addressed by the developer since he/she knows the details of the application domain and the input data used. EasyFJP automates the process of generating a parallel, tunable application “skeletons”, but it 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 or execution environment, rather than the other way around.

Likewise, the policy support discussed so far is not designed to automate application tuning, but to provide a framework that aims at capturing 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. More research is being done in this respect, as will be indicated later.

Moreover, an issue that may affect applicability is concerned with compatibility and interrelations with commonly-used techniques and libraries, such as multi-threading and AOP. In a broad sense, these techniques literally alter the ordinary semantics of a sequential application. Particularly, multi-threading 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 encapsulates 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 from this analysis is fork and joins 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, is 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 portion and the parallelized code of the application. Then, the component is used throughout the three steps. The classes that implement it are 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 to delegate such functionality.

Fig. 7 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 mostly part of the *Code Analyzer* component, although some code, the one related to inserts middleware API calls, belongs to the *Code Generator* component. To support SFJ and MFJ synchronization patterns, the previous algorithm is refined by extending the *PeerClassDirector* class and providing an extension to the *PeerClassBuilder* interface. *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 the peculiarities of the target parallel library (i.e., execution environment initialization), and also the library used to manipulate the input Java code.

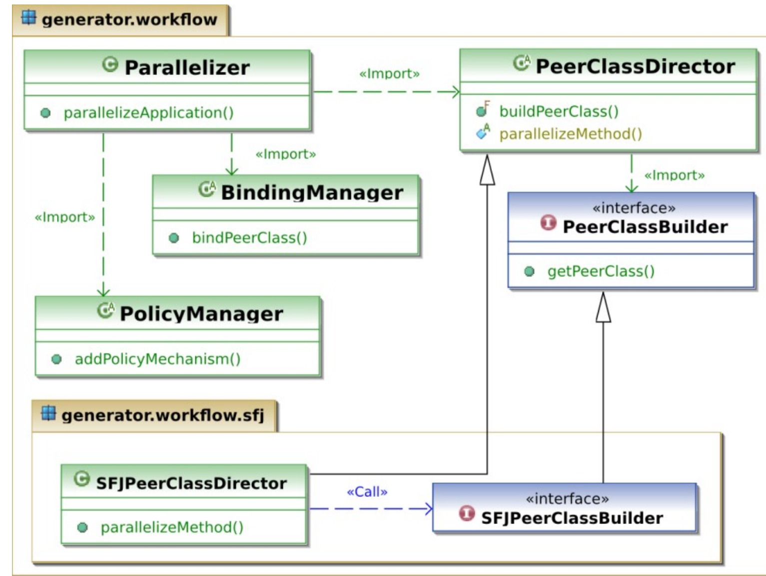

Fig. 7 EasyFJP main classes of the workflow package

5 Experimental evaluation

The practical implications of using EasyFJP are determined by two main aspects. One aspect is how competitive is implicitly supporting FJP synchronization patterns in D&C codes compared to explicit parallelism and classical parallel programming models. Another fundamental aspect is whether policies are effective to tune parallelized applications or not. Hence, we have conducted in the past experiments in the context of the MFJ synchronization pattern in (Mateos, Zunino, & Campo, 2010). Furthermore, for the sake of completeness, next we report experiments with SFJ through our new bindings to GridGain to further analyzing the trade-offs behind using EasyFJP.

As a testbed, we used 15 machines connected through a LAN with similar CPU capabilities running Ubuntu 11.04, Java 6 and GridGain 3.2.1. With the purpuse of simulates a more real Grid environment, where latency in the communication channels is greater than in a LAN network, the nodes were grouped into three-clusters. While the intra-cluster communication remained under the LAN conditions (100 Mbps), the communication between nodes placed in differents clusters (inter-cluster) were emulated with common WAN conditions. This means that for this type of links, and with the help of the software WANem 2.2[1], it was emulated a T1 connection type (bandwidth of 1,544 Mbps) with a round trip lattency of 160 ms and a jitter of 10 ms, resulting in inter-cluster communication latencies between 150-170 ms.

Regarding the application codes tested, it was used a ray tracing and a gene sequence alignment applications, 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.

Ray tracing (http://en.wikipedia.org/wiki/Ray_tracing_(graphics)) is a technique for generating an image by tracing the path of light through pixels in an image plane and simulating the effects of its encounters with virtual objects. The technique is capable of producing a very high degree of visual realism, usually higher than that of typical scanline rendering methods, but at a greater computational cost. Moreover, in bioinformatics, sequence alignment (http://en.wikipedia.org/wiki/Sequence_alignment) refers to a way of arranging the sequences of DNA, RNA or protein to identify regions of similarity that may be a consequence of functional, structural, or evolutionary relationships between the sequences. Sequence alignments are also used for non-biological sequences, such as those present in natural language or in financial data.

1 WANem (http://wanem.sourceforge.net/) is a software for emulating WAN conditions over a LAN

Fig. 8 Variables and values of SJF scenarios

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). As Fig. 8 shows, for ray tracing we used two scenes with two resolutions (1024x1024 and 2048x2048) that represent four different inputs of the application. In addition, three task granularities were used: 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, five databases with real disease information represented the application inputs and 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 (Lämmel, 2007). Hence, an escenario is represented by the four variables shown as columns in Fig. 8. The combinations of the values for each variable resulted in a total of 108 exercised scenarios (48 scenarios for ray tracing and 60 scenarios for the sequence alignment application).

Fig. 9 and Fig. 10 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 the case of the ray tracing application, 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 (Mateos, Zunino, & Campo, 2010).

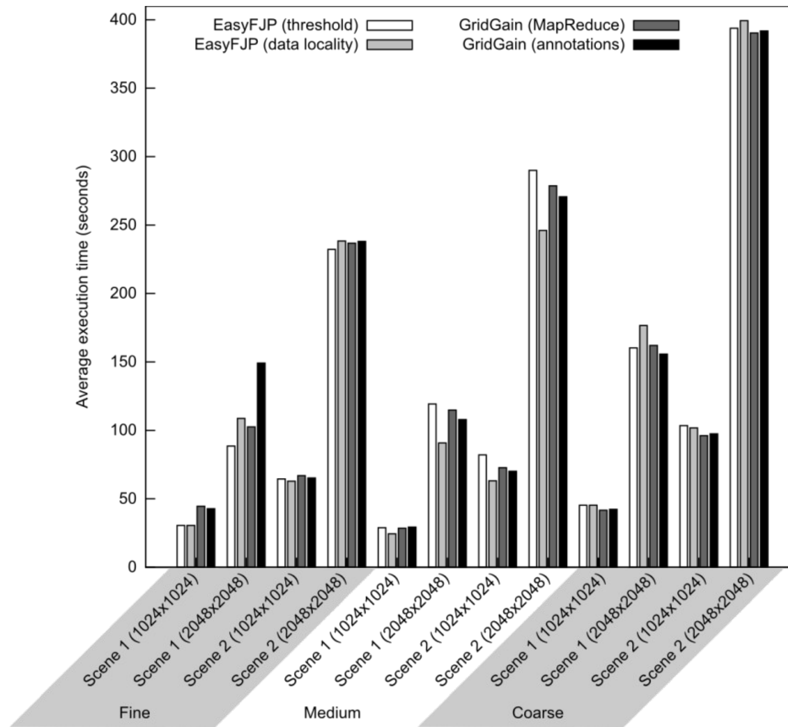

Fig. 9 Ray tracing application: Average execution time

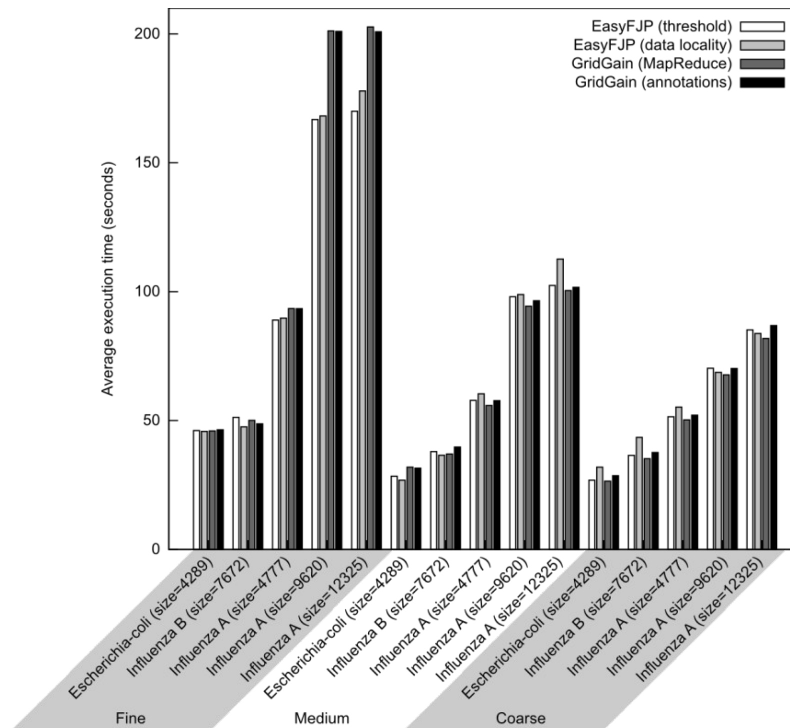

Fig. 10 Sequence alignment application: Average execution time

6 Conclusions

EasyFJP offers another 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 the latter. This separation, together with the simplicity of FJP and D&C, makes EasyFJP suitable for gradually introducing sequential programmers into parallel programming.

The experimental results shown in this paper and the ones reported in (Mateos, Zunino, & Campo, An Approach for Non-Intrusively Adding Malleable Fork/Join Parallelism into Ordinary JavaBean Compliant Applications, 2010) confirm that FJP-based implicit parallelism and policy-oriented explicit tuning, glued together via generative programming, are a viable approach to PaaC. Encouraging results were obtained for both fork-join synchronization patterns. 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, since only two parallel libraries (one supporting MFJ and another implementing SFJ) have been used. 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, which are a new execution environment characterized by computing resources simultaneously supporting high-levels of platform heterogeneity through virtualization technologies.

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 is known to improve programmer’s productivity (Dig, 2011). We are therefore building an IDE support to simplify the adoption and use of EasyFJP based on the Eclipse IDE for Java. Finally, we have produced 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 (Mateos, Zunino, Hirsch, & Fernández, 2012) and Java popularity is not that high compared to this scripting languages. At present, we have redesigned the EasyFJP policy API and its associated runtime support to allow users to code policies in Python and Groovy. Evaluating important aspect such as overhead (due to the inherent expensive nature of scripting languages) and usability is subject to further research.

References

Aldinucci, M., Danelutto, M., & Dazzi, P. (2007). Muskel: An Expandable Skeleton Environment. *Scalable Computing: Practice and Experience , 8* (4), 325-341.

Aleem, M., Prodan, R., & Fahringer, T. (2010). JavaSymphony: A Programming and Execution Environment for Parallel and Distributed Many-Core Architectures. *Lecture Notes in Computer Science , 6272*, 139-150.

Amedro, B., Caromel, D., Huet, F., & Bodnartchouk, V. (2008). Java ProActive vs. Fortran MPI: Looking at the Future of Parallel Java. *IPDPS'08* (págs. 1-7). IEEE.

Baduel, L., Baude, F., Caromel, D., Contes, A., Huet, F., Morel, M., y otros. (2006). Grid Computing: Software Environments and Tools. Springer.

Bull, J., & Kambites, M. (2000). JOMP-An OpenMP-like Interface for Java. *ACM Conference on Java Grande (JAVA '00), San Francisco, CA, USA* (págs. 44-53). New York, NY, USA: ACM Press.

Chan, H., Gallagher, A., Goundan, A., Au Yeung, Y., Keen, A., & Olsson, R. (2009). Generic Operations and Capabilities in the JR Concurrent Programming Language. *Computer Languages, Systems and Structures , 35* (3), 293-305.

Chandra, R., Dagum, L., Kohr, D., Maydan, D., McDonald, J., & Menon, R. (2000). *Parallel Programming in OpenMP.* San Francisco, CA, USA: Morgan-Kaufmann Publishers Inc.

Danaher, J., Lee, I., & Leiserson, C. (2006). Programming with Exceptions in JCilk. *Science of Computer Programming , 63* (2), 147-171.

Dig, D. (2011). A Refactoring Approach to Parallelism. *IEEE Software , 28* (1), 17-22.

Duarte, R., Mota, A., & Sampaio, A. (2011). Introducing Concurrency in Sequential Java Via Laws. *Information Processing Letters , 111* (3), 129-134.
Freeh, V. (1996). A Comparison of Implicit and Explicit Parallel Programming. *Journal of Parallel and Distributed Computing , 34* (1), 50-65.
Haustein, M., & Lohr, K.-P. (2006). JAC: Declarative Java Concurrency. *Concurrency and Computation: Practice and Experience , 18* (5), 519-546.
Jugravu, A., & Fahringer, T. (2005). JavaSymphony, A Programming Model for the Grid. *Future Generation Computer Systems , 21* (1), 239-246.
Kim, W., & Voss, M. (2011). Multicore Desktop Programming with Intel Threading Building Blocks. *IEEE Software , 28* (1), 23-31.
Lämmel, R. (2007). Google's MapReduce Programming Model --- Revisited. *Science of Computer Programming , 68* (3), 208-237.
Lea, D. (2005). The java.util.concurrent Synchronizer Framework. *Science of Computer Programming , 58* (3), 293-309.
Lee, E. (2006). The Problem with Threads. *Computer , 39* (5), 33-42.
Mateos, C., Zunino, A., & Campo, M. (2008). JGRIM: An Approach for Easy Gridification of Applications. *Future Generation Computer Systems , 24* (2), 99-118.
Mateos, C., Zunino, A., & Campo, M. (2010). An Approach for Non-Intrusively Adding Malleable Fork/Join Parallelism into Ordinary JavaBean Compliant Applications. *Computer Languages, Systems and Structures , 36* (3), 53-59.
Mateos, C., Zunino, A., & Campo, M. (2010b). On the Evaluation of Gridification Effort and Runtime Aspects of JGRIM Applications. *Future Generation Computer Systems , 26* (6), 797-819.
Mateos, C., Zunino, A., Hirsch, M., & Fernández, M. (2012). Enhancing the BYG Gridification Tool with State-of-the-art Grid Scheduling Mechanisms and Explicit Tuning Support. *Advances in Engineering Software , 43*, 27-43.
Nahavandipoor, V. (2011). *Concurrent Programming in Mac OS X and iOS: Unleash Multicore Performance with Grand Central Dispatch* . O'Reilly Media.
Shafi, A., Carpenter, B., Baker, M., & Hussain, A. (2009). A Comparative Study of Java and C Performance in Two Large-Scale Parallel Applications. *Concurrency and Computation: Practice and Experience , 21* (15), 1882-1906.
Sobral, J., & Proença, A. (2007). Enabling JaSkel Skeletons for Clusters and Computational Grids. *CLUSTER'07* (págs. 365-371). IEEE.
Sourceforge.net. (2009). Java Parallel Processing Framework. *Java Parallel Processing Framework* .
Systems, G. (2011). GridGain = Real Time Big Data. *GridGain = Real Time Big Data* .
Taboada, G. L., Ramos, S., Expósito, R. R., Touriño, J., & Doallo, R. (2011). Java in the High Performance Computing Arena: Research, Practice and Experience. *Science of Computer Programming* .
Van Nieuwpoort, R., Wrzesinska, G., Jacobs, C., & Bal, H. (2010). Satin: A High-level and Efficient Grid Programming Model. *ACM Transactions on Programming Languages and Systems , 32*, 9:1-9:39.
Zhang, B.-Y., Guang-Wen, Yang, & Zheng, W.-M. (2006). Jcluster: An Efficient Java Parallel Environment on a Large-scale Heterogeneous Cluster. *Concurrency and Computation: Practice and Experience , 18* (12), 1541-1557.
Zhang, H., Lee, J., & Guha, R. (2008). VCluster: A Thread-based Java Middleware for SMP and Heterogeneous Clusters with Thread Migration Support. *Software: Practice and Experience , 38* (10), 1049-1071.