An Approach for Non-Intrusively Adding Malleable Fork/Join Parallelism into Ordinary JavaBeans Compliant Applications. (C. Mateos, A. Zunino, M. Campo). Computer Languages, Systems and Structures. Elsevier Science. ISSN: 1477-8424. Ed.: R. S. Ledley and B. Kalyanasundaram. Vol. 36, Num. 3, pp 288-315. 2010. DOI: doi:10.1016/j.cl.2009.12.003

An Approach for Non-Intrusively Adding Malleable Fork/Join Parallelism into Ordinary JavaBeans Compliant Applications

Cristian Mateos*, Alejandro Zunino, Marcelo Campo

ISISTAN Research Institute. UNICEN University. Campus Universitario, Tandil (B7001BBO), Buenos Aires, Argentina. Tel.: +54 (2293) 439682 ext. 35. Fax.: +54 (2293) 439683
Also Consejo Nacional de Investigaciones Cientficas y Tcnicas (CONICET)

Abstract

Motivated by the advent of powerful hardware such as SMP machines and execution environments such as Grids, research in parallel programming has gained much attention within the Distributed Computing community. There is a substantial body of efforts in the form of parallel libraries and frameworks that supply developers with programming tools to exploit parallelism in their applications. Still, many of these efforts prioritize performance over other important characteristics such as code invasiveness, ease of use and independence of the underlying executing hardware/environment. In this paper, we present EasyFJP, a new approach for semi-automatically injecting parallelism into sequential Java applications that offers a convenient balance to these four aspects. EasyFJP is based upon the popular fork/join parallel pattern, and combines implicit, application-level parallelism with explicit, non-invasive application tuning. Experiments performed with several classic CPU-intensive benchmarks and a real-world application confirm that EasyFJP effectively addresses these problems while delivers very competitive performance.

Key words: Parallel computing, fork-join parallelism, implicit parallelism, non-invasive tuning, Java

1. Introduction

Fork/join parallelism (FJP) is a simple but effective design technique for parallelizing sequential applications [56]. FJP is based on expressing parallelism by means of two basic primitives: *fork*, which starts the execution of a code fragment –commonly a procedure or a method– in parallel, and *join*, which blocks the main application thread until the execution of these code fragments finishes. To handle the execution of forked code fragments in parallel, FJP-oriented libraries rely on specialized schedulers responsible for efficiently handling parallel subcomputations.

Particularly, FJP is suitable for parallelizing the family of divide and conquer algorithms. Divide and conquer applications solve problems by breaking them down into several subproblems of the same type, until trivial problems are obtained, which are solved directly. The solutions to the different subproblems are then combined to build the solution to the whole problem. Like divide and conquer algorithms, most FJP algorithms are recursive [56]: they repeatedly generate subtasks (i.e. forks) for each subproblem whose solutions are combined (i.e. join) to give a solution to the original problem. Small subproblems are commonly solved by calling a fragment of sequential code.

To some extent, FJP provides an alternative to the well-known thread programming model for parallelizing applications. This model has been receiving strong criticism [57] due to the complexity of programming, testing and debugging threads. In fact, an FJP framework is planned to be included in the next release (estimated in early 2010) of Java[1], which has offered threads as first-class citizens for many years. Intuitively, parallel but easy-to-use programming patterns like FJP are of major importance given the increasing availability of multi-core/multi-processor machines, so as to boost the performance of today's

*Corresponding author.
Email address: cmateos2006@gmail.com (Cristian Mateos)
[1] http://www.infoq.com/news/2007/07/concurrency-java-se-7

An Approach for Non-Intrusively Adding Malleable Fork/Join Parallelism into Ordinary JavaBeans Compliant Applications. (C. Mateos, A. Zunino, M. Campo). Computer Languages, Systems and Structures. Elsevier Science. ISSN: 1477-8424. Ed.: R. S. Ledley and B. Kalyanasundaram. Vol. 36, Num. 3, pp 288-315. 2010. DOI: doi:10.1016/j.cl.2009.12.003

sequential applications without the need for programmers with a solid background on parallel programming. In other words, FJP frameworks provide a convenient balance between delivered performance and ease of programming compared to multithread programming.

Noticeably, FJP is not only circumscribed to SMP machines, but can be also applied in other execution environments where the notions of task and processor exist, such as computer clusters [84]. In this context, tasks resulting from issuing forks can be executed in parallel on the machines of a cluster, thus potentially further increasing performance and scalability. More recently, Grid Computing [33, 34] has emerged as an exciting paradigm for parallel distributed computing. A Grid arranges the hardware resources from geographically dispersed sites to provide applications with vast amounts of computational resources. Interestingly, SMP machines, clusters and Grids alike can be used to execute FJP tasks, since these environments can be viewed in a general sense as a number of processing nodes (i.e. CPUs or individual machines) interconnected through a "network" infrastructure that provides communication capabilities (i.e. a system bus, a high-speed LAN or a WAN). This uniformity suggests that the same FJP application could be run in any of these environments, provided there is a specialized scheduler able to handle tasks according to the characteristics of the underlying execution hardware support. For example, a requirement for higher performance on a multi-core application may be fulfilled by modifying the application to exploit a Grid scheduler.

In dealing with the hardware and software diversity inherent to parallel environments, and specially Grids, Java has gained much popularity due to its "write once, run anywhere" property that promotes platform independence and the fact that its delivered performance is competitive with respect to that of conventional HPC languages [63]. However, historically, Java parallel libraries have focused on providing support for running applications on a particular parallel environment. Several tools for SMP (e.g. JOMP [15], Doug Lea's framework [56]), cluster (e.g. MPJ [16], jPVM [81], Hyperion [43]) and Grid programming (e.g. ProActive [10], JavaSymphony [50], Satin [84]) have been proposed. Basically, the aim of these tools is to supply developers with programming APIs and directives for starting and coordinating the execution of subtasks in parallel. However, this approach leads to source codes polluted with parallel instructions that depend on the library being used, compromising maintainability and portability to other libraries and execution environments. In other words, there is not a clear separation between the tasks of writing the application logic and parallelizing it. Besides, using these tools requires expertise on parallel programming. This intrusive approach to parallelism is also followed by several contemporary parallel languages designed to increase application programmer's productivity, such as Fortress (Sun) [6], X10 (IBM) [21], Axum [42] (Microsoft) and Chapel (Cray) [18]. However, whether these relatively new parallel languages will become widespread remains to be seen. In fact, many researchers promote the idea of extending for parallelism commonly employed languages rather than building parallel-specific languages from scratch. Examples of such dialects are UPC [31] (C), Intel® Threading Building Blocks (TBB) [69] (C++), Athapascan [37] (C/C++), and Titanium [86] (Java).

A crucial issue when introducing parallelism is to determine whether a sequential code will benefit from being parallelized or not. In particular, for FJP applications, small task granularities[2] may negatively affect speedup, as the cost of managing computations (e.g. creating and starting individual runtime tasks for them) may be greater than the computation times themselves. This is often avoided by using thresholds in the code to establish limits in the number of tasks injected into the runtime system, or *memoization*, this is, reusing results when subcomputations overlap. Again, existing Java parallel libraries follow an intrusive approach to tune the performance of parallelized applications, since these optimizations must be explicit in the application source code, which compromises maintainability and testability. Moreover, the optimizations introduced into an application may not be applicable when ported to a different execution environment, which leads to even more code modifications. There is, however, an increasing degree of consensus among the HPC community on the idea that for parallel applications not only performance is important, but also how well parallelism is abstracted and hidden from the application code [52].

This paper proposes EasyFJP, a new approach to mitigate these problems, which takes advantage of

[2]Throughout the rest of the paper, the term "granularity" should be understood as the computational requirements of the parallel runtime tasks resulting from parallelizing an application and should not be confused with the component granularity notion of component-based software systems.

An Approach for Non-Intrusively Adding Malleable Fork/Join Parallelism into Ordinary JavaBeans Compliant Applications. (C. Mateos, A. Zunino, M. Campo). Computer Languages, Systems and Structures. Elsevier Science. ISSN: 1477-8424. Ed.: R. S. Ledley and B. Kalyanasundaram. Vol. 36, Num. 3, pp 288-315. 2010. DOI: doi:10.1016/j.cl.2009.12.003

the implicit fork/join structure of sequential divide and conquer applications to generate a parallel version of the code that accesses a parallel API or library of the user's choice. A recursive call within the code is interpreted as a fork, whereas an instruction in which a result of a forked computation is read is interpreted as a join. Fork and join points are spotted through a generic algorithm that generates the actual parallel code subordinated to a particular parallel library. In addition, a fork point can be attached a *policy*, which is a user-provided rule that dynamically decides whether the fork takes place or the associated code is sequentially executed instead. Policies encapsulate the logic to efficiently execute applications by relying on an intuitive framework that captures common optimization heuristics for divide and conquer applications. Policies are non-invasively associated to the application code through an external configuration file, which is processed at runtime.

EasyFJP is targeted at parallelizing Java programs that follow a recursive (divide and conquer) structure, which is indeed applicable to a broad range of problems [58]. The contribution of this paper is a method for developing task-based Java parallel applications that (a) is based on application-level implicit FJP parallelism that does not require explicit usage of parallelism within the application code, (b) features integration with existing libraries and platforms for parallel and distributed development, and (c) offers a non-intrusive, rule-based mechanism to tune the same source code to various target execution environments. EasyFJP uses a generative programming approach to build FJP applications from sequential codes, and is materialized as a proof-of-concept tool that automatically outputs parallel code for a target parallel library including placeholders for attaching policies.

The rest of the paper is organized as follows. The next section discusses related works, and explains how EasyFJP complements them and improves over them. Section 3 overviews EasyFJP. Later, Section 4 briefly describes its implementation. After that, Section 5 reports an experimental evaluation of EasyFJP. Finally, Section 6 concludes the paper.

2. Background

The advent of sophisticated hardware and execution environments has motivated the development of many libraries and platforms for parallel programming. Particularly, we are interested in Java programming tools for implementing CPU-intensive applications based on FJP or similar embarrassingly parallel models, such as master-worker and bag-of-tasks [71]. Models exclusively based on explicit message passing and/or thread programming, or oriented towards data intensive applications are out of the scope of this section. Below we summarize these efforts.

With respect to parallelism on single machines, Javar [12] is a restructuring tool that paralellizes loops and recursive calls by converting them into multithreaded structures to run in parallel. Doug Lea's framework [56] is a set of classes (bundled into Java since version 5) that provides common functionality for managing synchronization state, blocking/unblocking of concurrent subcomputations and queuing. JCilk [25] extends the Java language with the *spawn* and *sync* fork/join primitives from the Cilk [14] multithreaded language. For each spawnable method, two different clones are created: a fast clone that executes in the common case where serial semantics suffice, and a slow clone that executes when parallel semantics are required. All communication due to scheduling is performed only when executing slow clones. This mechanism, which is prescribed by the computation model of the Cilk language, is known as the "two-clones" strategy [36]. Moreover, JCilk obeys the ordinary sequential semantics of the try/catch construct when executed on a single processor machine, but causes parallel computations to abort whenever an exception is thrown when executed on an SMP machine. JAC [44] simplifies concurrent programming by separating the application logic from thread declaration and synchronization through Java annotations. JAC emphasizes on removing the differences between sequential and concurrent code, promoting code reuse. In addition, a precompiler to translate from JAC to Doug Lea's framework exists [90]. Similarly, JOMP [15] is a Java implementation of OpenMP [20], a standard set of directives and library routines for shared memory parallel programming. JOMP provides a compiler and a runtime library that support a large subset of the OpenMP specification.

There are also several tools for Java-based parallel programming in distributed environments. JR [19] is a dialect of Java that provides a rich concurrency model supporting remote virtual machine/object creation and method invocation, asynchronous communication, rendezvous and dynamic process creation. JR

An Approach for Non-Intrusively Adding Malleable Fork/Join Parallelism into Ordinary JavaBeans Compliant Applications. (C. Mateos, A. Zunino, M. Campo). Computer Languages, Systems and Structures. Elsevier Science. ISSN: 1477-8424. Ed.: R. S. Ledley and B. Kalyanasundaram. Vol. 36, Num. 3, pp 288-315. 2010. DOI: doi:10.1016/j.cl.2009.12.003

programs are then translated into standard Java programs. JCluster [88] is a platform for executing task--based parallel applications in heterogeneous clusters. Subcomputations are scheduled according to a novel algorithm called transitive random stealing (TRS), which improves random stealing. JCluster also features PVM [38] and MPI [29] interfaces for implementing message-based parallel applications. PJ [51] provides an uniform API for loop parallelism and MPI-like message passing for cluster/SMP programming. Satin [84] is a library for parallelizing divide and conquer codes on LANs and WANs that follows the semantics of Cilk. Programmers mark through API classes and methods the operations that must be run in parallel. Then, Satin instruments the compiled code to execute the application in parallel on a Grid. JavaSymphony [50] is a performance-oriented platform with a semi-automatic execution model that transparently deals with migration, parallelism and load balancing of Grid applications, and at the same time allows programmers to control such features via API calls in the application code. Similarly, VCluster [89] supports execution of thread-based applications on SMP clusters. Threads can migrate between hosts for load balancing purposes and interchange data through certain *virtual channels* that are independent of the location of threads. Another related platform is Babylon [82], which support code mobility, parallelism and inter-thread communication in an uniform API.

Furthermore, ProActive [10, 54] is a platform for implementing object-oriented parallel mobile applications. A ProActive application is composed of mobile entities called *active objects*. Active objects serve methods calls originated from other (active) objects, and call methods implemented by other local or remote (active) objects. Method calls are asynchronously handled through the *wait-by-necessity* mechanism, which is equivalent to future objects in Java. Active object creation, parallelism and mobility must be specified in the application code through API calls. JGRIM [64] is a method for Grid-enabling ordinary component-based applications. JGRIM focuses on non-invasively injecting existing Grid middleware services such as component brokering, mobility, and parallelism into sequential codes via Dependency Injection techniques [49]. Moreover, the Grid Component Model (GCM) [23, 3] of the CoreGRID NoE defines a model for creating component-based Grid applications, by which a component's behavior can be seamlessly attached extra-functional behavior related to distribution and parallelism. At present, there is an open source reference implementation of GCM called GridCOMP [47], which materializes this model on top of the ProActive platform and lets developers to establish the non-functional behavior of components via external pluggable rules [3]. Unlike EasyFJP, the component models prescribed by JGRIM and GCM aim at providing higher levels of composability and interoperability of user-level and platform-level components by promoting the convergence of Grid and SOA [11] concepts. Finally, with respect to the plethora of open source Java APIs for executing bag-of-tasks distributed applications, some examples are JPPF [74], GridGain [41], Xgrid [46] and JCGrid [73].

From a programming language perspective, the approaches to parallelism can be classified into implicit and explicit [35]. On one hand, implicit parallelism allows programmers to write their programs without any concern about the exploitation of parallelism, which is instead automatically performed by the compiler or the runtime system. On the other hand, explicit parallelism aims at supplying constructs or APIs for describing and coordinating parallel computations. Programmers have more control over the parallel execution, thus it is feasible to fully exploit parallelism to implement very efficient applications. However, explicit parallelism is difficult to deal with [35], since the burden of initiating, stopping and synchronizing parallel computations is placed on the programmer.

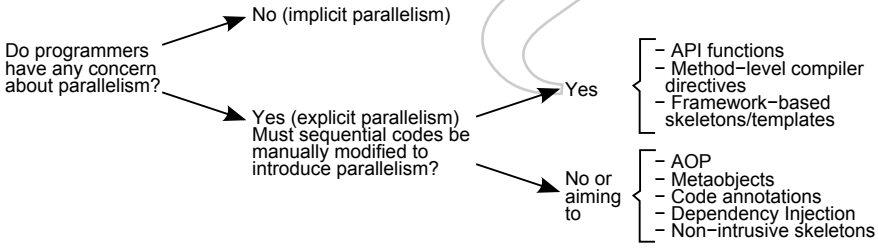


Figure 1: Approaches to parallelization in Java: A taxonomy

Although designed with simplicity in mind, many of the above efforts fall into the category of tools

An Approach for Non-Intrusively Adding Malleable Fork/Join Parallelism into Ordinary JavaBeans Compliant Applications. (C. Mateos, A. Zunino, M. Campo). Computer Languages, Systems and Structures. Elsevier Science. ISSN: 1477-8424. Ed.: R. S. Ledley and B. Kalyanasundaram. Vol. 36, Num. 3, pp 288-315. 2010. DOI: doi:10.1016/j.cl.2009.12.003

based on explicit parallelism. This has two clear disadvantages. First, parallelizing an application requires learning the parallel API of the tool being used, which may be difficult for an average programmer. Second, from a software engineering perspective, the resulting code is difficult to maintain and to port to other frameworks and libraries. Moreover, explicit parallelism usually leads to code that contains not only the boilerplate instructions for creating and coordinating subcomputations but also statements for tuning the execution of the subcomputations according to the nature of the application and the environment where it executes. This potentially makes the tuning rules invalid as soon as the application is ported to a different environment, for example, from a local cluster to a wide-area cluster.

An alternative solution to conventional explicit parallelism is to treat parallelism as a concern and thus to avoid mixing the logic of applications with the code that implements its parallel behavior (see Figure 1). This idea has been gaining momentum as shown by existing tools which are partially or entirely based on mechanisms for separation of concerns such as code annotations (JAC, Satin, GridGain), metaobjects (ProActive), Dependency Injection [49] (JGRIM) and dynamically pluggable rules (GridCOMP). Moreover, some efforts [17, 87, 24, 13] have supported the same idea through the application of AOP [53] to seamlessly combine sequential code (i.e. application logic) with code in charge of parallelism and application tuning (i.e. aspects). Similarly, other frameworks have proposed the use of skeletons, which capture recurring parallel programming patterns such as pipes, master-worker and farms without ideally affecting applications. Roughly, the modeled patterns are instantiated by programmers through the creation of the involved interacting components by either wrapping existing sequential applications (e.g. Muskel [5, 4]) or the specialization of framework classes (e.g. JaSkel [72], CO_2P_3S [59]). All in all, the key problems of the existing approaches to parallelism pursuing separation of concerns relate to:

- Applicability: Naturally, approaches designed to exploit single machines are not applicable to cluster and Grid settings. Conversely, many approaches designed to take advantage of such settings experience overheads due to the distributed nature of the platforms underneath [1], which makes them potentially less efficient when exploiting, for example, SMP machines. These problems violate the "handling heterogeneity" principle for parallel tools introduced in [26], which states that parallel applications should be easy to port to different parallel execution environments.

- Code intrusiveness: Approaches based on code annotations require explicit source code modifications to introduce task parallelism and application-specific optimizations, thus the resulting code is somewhat more difficult to handle thereafter from a software engineering standpoint. Metaobjects and specially AOP techniques have proven to be effective techniques in avoiding code modifications when introducing parallelism [27], but at the expense of demanding developers to learn and use another programming paradigm.

- Expertise: Roughly, approaches aimed at providing support for various parallel patterns and templates feature good applicability with respect to the range of applications that can be parallelized, but employing these approaches require a solid knowledge on parallel programming from developers. In addition, with these tools, the code structure of a sequential application is usually very different to that of its parallel counterpart. Therefore, introducing modifications to the application logic after parallelization unavoidably demands first to understand the produced parallel code. In opposition, by restricting the offered parallel patterns just to FJP, one would nevertheless provide a model for parallelizing a broad range of applications while not incurring too much in such structural difference.

In this sense, we exploit the concept of separation of concerns to provide a hybrid approach to develop parallel applications that *implicitly* introduces parallel behavior into existing sequential applications, and (optionally) allows programmers to *explicitly* tune the resulting parallel code by means of an uniform API without affecting the application logic. The goal is to get the best from both worlds: the simplicity of implicit parallelism [35], and both the flexibility and efficiency of explicit parallelism [35]. The execution and coordination of parallel applications are performed by transparently leveraging existing parallel libraries for cluster/Grid and SMP programming. Then, EasyFJP is suited for developers with limited knowledge on parallel programming.

In our view, EasyFJP does not compete but complements existing work by offering developers who are not experienced in parallel programming a versatile and effective tool for easily parallelizing applications.

An Approach for Non-Intrusively Adding Malleable Fork/Join Parallelism into Ordinary JavaBeans Compliant Applications. (C. Mateos, A. Zunino, M. Campo). Computer Languages, Systems and Structures. Elsevier Science. ISSN: 1477-8424. Ed.: R. S. Ledley and B. Kalyanasundaram. Vol. 36, Num. 3, pp 288-315. 2010. DOI: doi:10.1016/j.cl.2009.12.003

This is achieved by circumscribing parallelism to the popular FJP model and using a generative programming approach to automatically build parallel code that uses existing parallel libraries. Besides, developers who are proficient in parallel programming can manually optimize the code generated by our tool. The next section describes EasyFJP.

3. Approach

EasyFJP is a new approach for simplifying the parallelization of sequential Java applications. EasyFJP differs from similar efforts in that it offers a convenient balance to the dimensions of applicability, code intrusiveness and expertise discussed above. First, broad applicability is achieved by targeting the Java language and the FJP parallel model, and leveraging the services of existing parallel libraries. Second, low intrusiveness is achieved by using a generative programming approach to generate parallel code from sequential code, and keeping the tuning behavior away from the generated parallel application. Precisely, this separation, together with the simplicity of FJP, makes EasyFJP suitable for developers with little expertise on parallel programming.

EasyFJP represents an alternative approach to simplify the development of parallel programs in the form of an "amalgam" that combines several good ideas already present in existing tools but not simultaneously exploited, namely:

- Wide applicability, to allow developers to run in parallel a broad range of sequential applications in various parallel environments such as multi-core machines and local/wide-area clusters.
- Flexibility and extensibility, to seamlessly exploit the scheduling and the synchronization capabilities featured by existing multi-core and distributed parallel libraries.
- Little intrusiveness, to minimize the impact of introducing parallelism into a sequential application.
- Use of standard Java and a simplified parallel programming model, thus novice developers can take advantage of parallelism without using a parallel dialect of Java or being proficient in parallel concepts.
- Malleability, to allow experienced parallel developers to tune the parallel applications resulting from using EasyFJP according to both the nature of the applications and the environments where they execute.

At the heart of EasyFJP is a semi-automatic parallelization process to introduce parallelism into applications. EasyFJP accepts as input ordinary sequential divide and conquer Java applications, and generates library-dependent parallel applications with hooks for incorporating application-specific and environment-specific optimizations. As depicted in Figure 2, this process comprises three steps, which conceptually operate as follows:

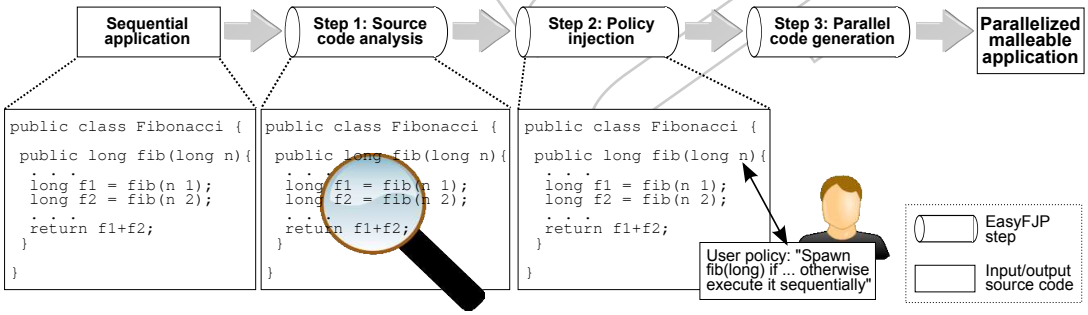


Figure 2: An overview of EasyFJP

An Approach for Non-Intrusively Adding Malleable Fork/Join Parallelism into Ordinary JavaBeans Compliant Applications. (C. Mateos, A. Zunino, M. Campo). Computer Languages, Systems and Structures. Elsevier Science. ISSN: 1477-8424. Ed.: R. S. Ledley and B. Kalyanasundaram. Vol. 36, Num. 3, pp 288-315. 2010. DOI: doi:10.1016/j.cl.2009.12.003

- *Source code analysis* (Section 3.1): The source code of the input sequential divide and conquer application is analyzed in order to spot the points in which recursive calls are performed and the results of such calls are accessed. These points are interpreted by EasyFJP as implicit forks and joins, respectively. Before feeding our analyzer, the programmer has to assign the results of recursive calls to local variables declared at the beginning of methods in his sequential application. Note that this is a simple code convention that does not involve API calls. A similar convention is also required by parallel frameworks such as Satin.

  In general, existing parallel libraries provide means to launch the execution in parallel of a single fork or a list of forks. Therefore, the parallelization of individual spotted forks can be straightforwardly performed at step 3 by replacing each fork by the necessary API code to individually execute each fork in parallel. However, there are semantic differences among parallel libraries with respect to the primitives offered to express join points. Operationally, join primitives can be grouped into two types: single-fork and multi-fork. The former models one-to-one relationships between fork and join points, this is, for every fork a join call must be issued within the code to wait for the result of a particular fork. Conversely, multi-fork joins only allow the application to wait for the results of the forks previously issued up to the join call. For example, in the application code of Figure 2 (step 1), a multi-fork join before the return sentence would cause the application to block until *both* f_1 and f_2 are available, whereas two single-fork joins would be necessary to obtain the same behavior.

  Parallel libraries supporting single-fork joins (e.g. GridGain, JPPF) greatly simplify the task of automatically inserting library code to handle join points that is carried out at step 3, since each individual join can be trivially replaced by the corresponding parallel API call. However, multi-fork join primitives such as the one provided by Satin make this task more difficult, as it is necessary to perform a smarter analysis of the code to find proper places to insert synchronization by taking into account aspects such as the structure of sentences, variable scopes, etc. To this end, EasyFJP includes a library-independent code analysis algorithm, which is explained in Section 3.1.

- *Policy injection* (Section 3.2): The policy support is a non-intrusive mechanism by which programmers can customize for efficiency purposes the way a parallel application behaves at runtime. A policy is a user-supplied class that dynamically decides whether to actually fork a recursive call or execute it sequentially instead. For example, the Fibonacci application (Figure 2) could be instructed to fork a call to fib provided the depth within the execution tree is below some given threshold. Moreover, policies are non-invasively associated to fork points through an external configuration file. Then, these configured policies, which are basically the rules that control the amount of parallelism of an application, can be changed without modifying the application code.

  Interestingly, this approach to tuning allows developers to adjust parallelism according to the nature of their applications (e.g. using thresholds or memoization) as well as the dynamics and characteristics of the underlying execution environment (e.g. avoiding too many forks with large parameters in a high-latency network) without affecting the application logic. For building policies, EasyFJP offers a profiling API for obtaining runtime information about both the running application and the execution infrastructure. The usage of policies is not mandatory for parallelizing applications. In addition, the separation promoted by this mechanism between the tasks of writing application logic and tuning it contributes to the application development process, as these two groups of tasks can be carried out independently by programmers with different skills. For example, policy coding could be performed by a programmer proficient in parallel concepts and our policy API.

- *Parallel code generation* (Section 3.3): This step involves the generation of the parallel code that depends on the parallel library selected by the developer. To this end, the process prescribes the existence of *builders*, which are library-dependent components that comprise the necessary functionality to adapt a sequential application to the application structure of the target parallel library. For example, some libraries require applications to extend from certain API-classes, include default constructors, and so on. Besides, builders are in charge of taking advantage of the target library to process fork and joins from the analysis at step 1, and to inject the glue code to invoke the user policies defined at step 2.

An Approach for Non-Intrusively Adding Malleable Fork/Join Parallelism into Ordinary JavaBeans Compliant Applications. (C. Mateos, A. Zunino, M. Campo). Computer Languages, Systems and Structures. Elsevier Science. ISSN: 1477-8424. Ed.: R. S. Ledley and B. Kalyanasundaram. Vol. 36, Num. 3, pp 288-315. 2010. DOI: doi:10.1016/j.cl.2009.12.003

At present, we have implemented builders for Satin [84], but bindings to other parallel libraries are underway. We are for example working on builders for Doug Lea's framework [56] for multi-core programming and the GridGain [41] distributed platform. With regard to fork processing, developing builders for libraries based on FJP such as Satin mainly involves translation, this is, recursive methods in the input FJP application are forked in the output parallel application via appropriate calls to the target library API. However, supporting libraries that rely on execution models such as master--worker or bag-of-tasks (e.g. GridGain), in which there are no hierarchical relationships between parallel tasks, is not straightforward since builders must emulate FJP by using the non-FJP primitives of the underlying parallel library. With respect to join processing, libraries based on multi-fork join primitives such as Satin makes code generation significantly more challenging than those based on single-fork join primitives, as it is necessary to deeply analyze the input application to introduce a minimal number of calls to such primitives.

It is worth noting that this parallelization process does not aim at introducing parallelism into any kind of Java application, but targets component-based Java applications only. Component-based programming allows developers to implement applications comprising logical components with well-defined interfaces [78]. Besides, components are designed to not share state and to communicate with other components through loosely-coupled operation calls. This leads to decoupled building blocks where any interaction that involves tightly-coupled communication is disallowed, such as invoking component operations by passing parameters by reference. This, in turn, allows communicating components to be executed in different address spaces without the need of explicitly using more coupled communication mechanisms such as shared objects. In fact, component-based notions are being extensively applied in distributed programming, as evidenced by proposed Grid component models such as the GCM [23, 3] and the K-Components [30]. Also, in Java, component-based programming is commonplace [64], given by the high popularity of component models for Java such as JavaBeans [77] and Dependency Injection [49]. For these reasons, we believe the applicability of EasyFJP is not compromised.

Precisely, the development model promoted by EasyFJP is based upon the JavaBeans [77] specification, which states among other things that any individual application class must (a) be serializable, and (b) contain getters and setters to access its properties. In this context, a component is just an ordinary class that is implemented by following a number of coding and design conventions. Particularly, (a) is crucial to our work as it ensures that application objects can be transparently migrated across machines when using parallelism while targeting a distributed environment. On the other hand, (b) defines a convention to read and to set the properties of a component from the outside. The utility of this latter code convention for our work will be discussed in Section 3.3.

The source code conventions imposed by EasyFJP are very simple to follow. Specifically, before processing an application with EasyFJP, the developer must format its source code by following the coding conventions discussed so far, namely adhering to the JavaBeans specification and storing results of recursive calls in local variables. On one hand, JavaBeans were originally defined as reusable software components that can be manipulated (e.g. created and composed) by means of graphical development tools [77]. Note that most current IDEs for Java such as Eclipse allow developers to automatically refactor ordinary classes to be compliant to the JavaBeans model (e.g. associating setters/getters to the instance variables of a class) or to change the shape of the code implementing their methods. In this sense, using these conventions does not mean greater development effort, because they are feasible to be automated by exploiting IDE refactoring tools. For example, we are developing a plug-in for Eclipse to support the parallelization process of EasyFJP, which as mentioned above already include tools for performing such kind of code transformations.

Let us illustrate the parallelization process of EasyFJP with an example application. For instance, if we have a class including a recursive solution to compute the n^{th} Fibonacci number like:

```
public class Fibonacci{
  private Hashtable preComputedValues = new Hashtable();
  public long fib(long n){
    if (n < 2)
      return n;
    if (preComputedValues.containsKey(n))
      return preComputedValues.get(n);
```

An Approach for Non-Intrusively Adding Malleable Fork/Join Parallelism into Ordinary JavaBeans Compliant Applications. (C. Mateos, A. Zunino, M. Campo). Computer Languages, Systems and Structures. Elsevier Science. ISSN: 1477-8424. Ed.: R. S. Ledley and B. Kalyanasundaram. Vol. 36, Num. 3, pp 288-315. 2010. DOI: doi:10.1016/j.cl.2009.12.003

```
    return fib(n - 1) + fib(n - 2);
  }
  // Other methods
  public static void main(String[] args){...}
}
```

, formatting the above application results in the following code:

```
1  public class Fibonacci implements Serializable{
2    private Hashtable preComputedValues = null;
3    public long fib(long n){
4      long f1 = 0; long f2 = 0;
5      if (n < 2)
6        return n;
7      if (getPreComputedValues().containsKey(n))
8        return getPreComputedValues().get(n);
9      f1 = fib(n - 1);
10     f2 = fib(n - 2);
11     return f1 + f2;
12   }
13   public Hashtable getPreComputedValues(){
14     return preComputedValues;
15   }
16   public Hashtable setPreComputedValues(Hashtable preComputedValues){
17     this.preComputedValues = preComputedValues;
18   }
19   // Other methods
20   public static void main(String[] args){...}
21 }
```

Basically, lines 9-10 were added to store the results of both recursive calls into two local variables f1 and f2 declared at the beginning of the recursive method (line 4). Besides, we made the class serializable (line 1) and added getters and setters for its preComputedValues instance variable (lines 13-18), which must be used within the code to properly access the variable (lines 7-8). Once these conventions are applied, the modified source code along with some configuration are processed with EasyFJP to obtain the parallel counterpart of the sequential application. Specifically, this configuration is an XML file that specifies which classes (fully qualified names) and methods from these classes must be analyzed for introducing parallelism. In our example, the configuration would be:

```
1  <application name="Fibonacci"
2      xsi:noNamespaceSchemaLocation="configuration.xsd"
3      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
4    <components>
5      <component id="Fibonacci" class="Fibonacci">
6        <method id="fib" name="fib">
7          <parameter type="long"/>
8        </method>
9      </component>
10     ...
11   </components>
12   <policies>
13     ...
14   </policies>
15 </application>
```

The components element specifies the application classes and methods that will be parallelized. It is also possible to parallelize several methods from an individual class. Finally, the policies element includes the optimizations associated to the method(s) to be parallelized, which are explained in Section 3.2.

3.1. Step 1: Source code analysis

Before feeding EasyFJP with a sequential application, the results of recursive methods must be stored in local variables. Roughly, this convention allows EasyFJP to automatically identify implicit fork sentences and spot the points in which synchronization barriers are needed, so as to ensure that recursive results are always available before they are accessed. In case a programmer targets a library including a single-fork

An Approach for Non-Intrusively Adding Malleable Fork/Join Parallelism into Ordinary JavaBeans Compliant Applications. (C. Mateos, A. Zunino, M. Campo). Computer Languages, Systems and Structures. Elsevier Science. ISSN: 1477-8424. Ed.: R. S. Ledley and B. Kalyanasundaram. Vol. 36, Num. 3, pp 288-315. 2010. DOI: doi:10.1016/j.cl.2009.12.003

join primitive such as Doug Lea's framework or GridGain, the resulting joint points are equivalent to the points in which those local variables are read. However, when generating code for a parallel library based on a multi-fork join primitive such as Satin, a smarter code analysis is necessary to minimize the number of inserted synchronization barriers.

A naive solution to the problem of automatically inserting synchronization is to blindly add a barrier right before any access to a local variable representing a recursive result. However, this solution generates more calls to the underlying join primitive than needed, and depending on the library being used and the cost of calling the primitive, this may negatively affect the performance of the resulting application. Therefore, we designed an heuristic that aims at inserting a minimal number of synchronization barriers and at the same time preserving the semantics of the original application. The algorithm works by walking through the instructions of a method and detecting the points in which a local variable is either *defined* or *used* by a sentence. A local variable is defined and thus becomes a *parallel variable* when the result of a recursive method is assigned to it. On the contrary, a parallel variable is used when its value is read. When executing in parallel, to work properly recursive methods can read parallel variables provided a join has been previously invoked. Based on the identified join points at this step, EasyFJP modifies the source code so as to ensure that a library-specific join primitive is called between the definition and use of any parallel variable, for any execution path between these two points. When targeting a parallel library based on single-fork join primitives, the analysis simply outputs the points in which a parallel variable is used. However, when targeting a library relying on multi-fork join primitives, the analysis employs an heuristic algorithm to keep the correctness of the resulting parallel application while minimizing the identified join points. Any regular local variable that does not represent results from parallel computations (i.e. non--parallel) is naturally ignored by the algorithm.

Algorithm 1 Spotting multi-fork join points

```
procedure IDENTIFYSYNCPOINTS(rootScope)
   syncPoints ← empty
   for all sentence ∈ TRAVERSEDEPTHFIRST(rootScope) do
      if varName ← ISPARALLELVAR(sentence) then
         currentScope ← GETSCOPE(sentence)
         if BEINGUSED(varName,sentence) = true then
            if GETFIRSTSTATE(varName,currentScope) = UNSAFE then
               SYNCVARSINSCOPE(currentScope)
               ADDELEMENT(syncPoints,sentence)
            end if
         else if BEINGDEFINED(varName,sentence) = true then
            DESYNCVARUPTOROOT(varName,currentScope)
         end if
      end if
   end for
   return syncPoints
end procedure
```

Algorithm 1 summarizes the process of identifying the multi-fork join points (*syncPoints*) of a divide and conquer method based on its associated tree-based representation. Basically, the nodes of the tree represent the different scopes of the method, this is, the root scope given by the method itself and the scopes resulting from container sentences (e.g. conditionals, loops, etc). The arcs of the three represent the relationships between the scopes.

The algorithm traverses the sentences of the tree in a depth-first fashion looking for definitions and uses of parallel variables. To this end, the algorithm maintains a map with the parallel variables and their associated state per scope. Possible states are SAFE (up to the current analyzed instruction the variable is safe to use; a synchronization barrier is not needed) and UNSAFE (unsafe to use; a barrier from where the variable is defined is needed). The algorithm takes into account the scope at which parallel variables are defined and used, this is, it computes the state of each variable according to the state it has within the

An Approach for Non-Intrusively Adding Malleable Fork/Join Parallelism into Ordinary JavaBeans Compliant Applications. (C. Mateos, A. Zunino, M. Campo). Computer Languages, Systems and Structures. Elsevier Science. ISSN: 1477-8424. Ed.: R. S. Ledley and B. Kalyanasundaram. Vol. 36, Num. 3, pp 288-315. 2010. DOI: doi:10.1016/j.cl.2009.12.003

(scope) node of the tree where the variable is read and the state of the same variable within the ancestors of that node. The algorithm uses several helper functions, which are listed and summarized in Table 1.

Table 1: Helper functions of the algorithm for identifying multi-fork join points

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

Let us apply the algorithm to the sequential method shown below. The method contains one non--parallel variable (nonParallelVar) and two parallel variables (varA and varB). The points in which a call to a multi-join barrier are needed are explicitly indicated in the source code. Figure 3 depicts the state of varA and varB within their associated scopes as the analysis progresses. It is worth mentioning that the method does not implement any useful computation but it will be enough for illustration purposes.

```
1  public String recursiveMethod() { // Scope 1
2    ...
3    boolean nonParallelVar = (Math.random() > 0.5) ? true : false;
4    String varA = recursiveMethod();
5    if (!nonParallelVar) { // Scope 1.1
6      String varB = recursiveMethod();
7      if (Math.random() > 0.5) { // Block 1.1.1
8        // A multi-fork join should be issued here
9        System.out.println(varB);
10       varA = recursiveMethod();
11     }
12   }
13   if (nonParallelVar) { // Scope 1.2
14     // A multi-fork join should be issued here too
15     System.out.println(varA);
16   }
17   ...
18 }
```

The algorithm iterates the instructions up to line 4, in which varA is defined. Hence, varA becomes UNSAFE in scope 1 (see Figure 3 (a)). At line 6, varB is defined within scope 1.1, which makes it UNSAFE in scope 1.1 and its parent scope 1 (see Figure 3 (b)). At line 9, varB is used within scope 1.1.1. Its first occurrence is encountered in the parent of scope 1.1.1 as UNSAFE. All parallel variables in the variable maps of scope 1.1.1 (none) and its ancestors (varA and varB) are set to SAFE in scope 1.1.1, and the line right before line 9 is regarded as a multi-fork join point (Figure 3 (c)). At line 10, another definition of varA

An Approach for Non-Intrusively Adding Malleable Fork/Join Parallelism into Ordinary JavaBeans Compliant Applications. (C. Mateos, A. Zunino, M. Campo). Computer Languages, Systems and Structures. Elsevier Science. ISSN: 1477-8424. Ed.: R. S. Ledley and B. Kalyanasundaram. Vol. 36, Num. 3, pp 288-315. 2010. DOI: doi:10.1016/j.cl.2009.12.003

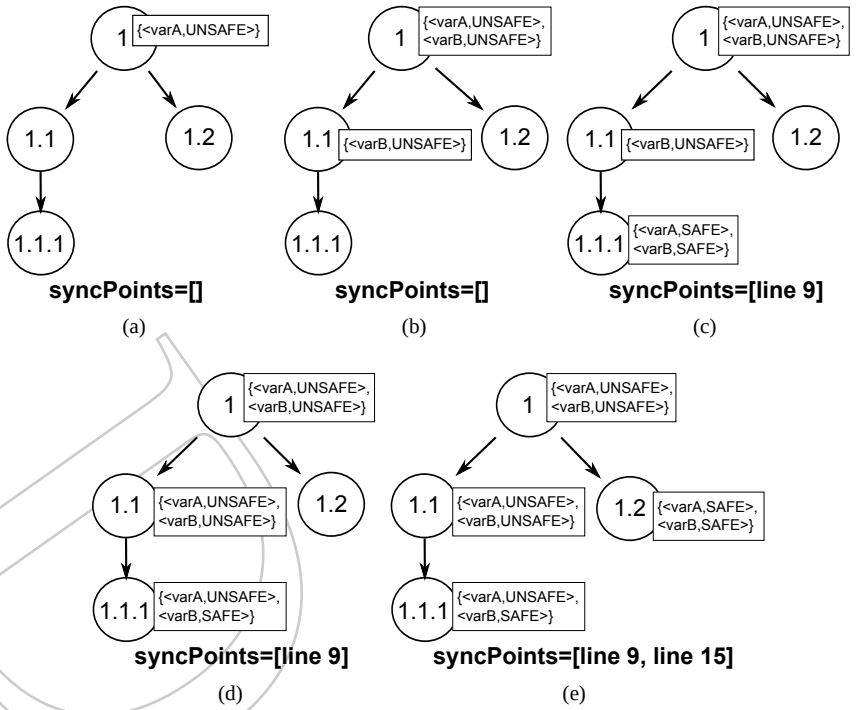


Figure 3: Contents of the variable maps of the example method in the different steps of the algorithm

is found, which makes the variable UNSAFE in scopes 1.1.1, 1.1 and 1 (Figure 3 (d)). At line 15, varA is being used within scope 1.2. According to its parent scope 1, the first state of this variable is UNSAFE. This causes to set to SAFE in scope 1.2 all variables found in the maps of scope 1.2 (none) and its ancestors (varA and varB), and to regard the line right before line 15 as a multi-fork join point (Figure 3 (e)).

Inserting a call to a multi-fork join primitive right before the spotted points *syncPoints* (i.e. lines 9 and 15) provides proper synchronization and guarantees the operational semantics of the sequential code. As shown, the algorithm minimizes the identified points that are passed on to the code generator of Section 3.3 to gain efficiency. Similarly, the algorithm performs a best-effort analysis to place such points optimally . A common case of optimization is when inserting synchronization within loops. For example, in the following code:

```
int[] varA = recursiveMethod(...);
...
for (int i=0; i < varA.length; i++){
  System.out.println(varA[i]);
}
```

, the line of the first use of the parallel variable (i.e. when the i_{th} element of varA is accessed) is not considered as the multi-fork join point –as the above unoptimized algorithm would do– but the line corresponding to the header of the loop. In consequence, the resulting code performs only one invocation to the join primitive instead of varA.length calls.

3.2. Step 2: Policy injection

In a broad sense, policies represent a mechanism that allows developers to express, separately from the application logic, customized strategies for applications to achieve better performance [62]. Policies have been widely employed in diverse areas such as mobile distributed computing [66], migratory Web agents [62], and Grid development [64]. Conceptually, a policy implements a user-specified rule that governs the behavior of an application within the underlying execution environment. Unlike approaches aimed at automatically tuning parallel applications [61, 67, 7], policies are oriented towards providing a flexible programmatic support to specify tuning decisions. Particularly, EasyFJP provides a policy-inspired tuning

An Approach for Non-Intrusively Adding Malleable Fork/Join Parallelism into Ordinary JavaBeans Compliant Applications. (C. Mateos, A. Zunino, M. Campo). Computer Languages, Systems and Structures. Elsevier Science. ISSN: 1477-8424. Ed.: R. S. Ledley and B. Kalyanasundaram. Vol. 36, Num. 3, pp 288-315. 2010. DOI: doi:10.1016/j.cl.2009.12.003

support that let developers to introduce common FJP optimization heuristics without altering application code by means of special Java classes.

Policies considered by EasyFJP fall into two types: application-specific and environment-specific. The former represents tuning decisions that are more affected by the algorithmic nature of the application being parallelized, whereas the latter represents optimization rules that aim at adjusting the level of parallelism of an application according to the capabilities of the environment where the application executes. On one hand, application-specific policies model the notions of threshold and memoization. On the other hand, environment-specific policies use runtime information provided by the environment (CPU and memory availability, network conditions, topology, etc.) to make their decisions. There is not, however, a clear line between these two types of policies as programmers can implement hybrid policies that combine threshold and/or memoization techniques together with environmental conditions. For example, the amount of memoized results for a memory-intensive application may be controlled by also taking into account the available memory in the executing cluster.

Threshold policies are employed to avoid forking a method more than needed and otherwise execute the method sequentially. For example, in the Fibonacci application, we may want to put a limit on the number of forks that are injected into the runtime system depending on the depth of the execution tree associated to the method at runtime. This decision is indicated to EasyFJP by associating the following policy to the fib method:

```
import easyFJP.policy.Policy;
import easyFJP.policy.ExecutionContext;

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

The code implements the Policy interface from the EasyFJP policy API and allows each execution of fib to be forked provided the current depth of the execution tree associated to the method is below 10. This depth is encapsulated in an ExecutionContext object, which additionally provides operations to further introspect the execution of the application, namely obtaining the values of method parameters. For example, for a recursive binary search method over an array, a similar policy may be associated to restrict parallelism depending on the size of the input array. Assuming the signature of the method is search(int element, int[] array), the policy would be:

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

The above code uses the execution context object to access the value of the second argument of each call to search to decide whether the size of the received array is large enough to justify a fork. Now, recall the structure of the configuration file that specifies the policies for an application. Then, for example, to attach the above threshold policy to the Fibonacci application, we must add the following declaration within the policies element:

```
...
<policies>
  <policy id="myThreshold" class="MyThresholdPolicy">
    <!-- Instances of the same policy can be attached
         to methods from different components -->
    <actUpon componentId="Fibonacci" methodId="fib"/>
  </policy>
</policies>
...
```

Memoization is another common optimization technique used to gain efficiency by having applications to avoid forking a method in case its associated result has been already computed and stored into a cache. In

An Approach for Non-Intrusively Adding Malleable Fork/Join Parallelism into Ordinary JavaBeans Compliant Applications. (C. Mateos, A. Zunino, M. Campo). Computer Languages, Systems and Structures. Elsevier Science. ISSN: 1477-8424. Ed.: R. S. Ledley and B. Kalyanasundaram. Vol. 36, Num. 3, pp 288-315. 2010. DOI: doi:10.1016/j.cl.2009.12.003

this sense, in our Fibonacci application, we may want to avoid recalculating previously computed results, as the nature of the application makes subcomputations to overlap. From a programmer's perspective, coding a memoization policy requires deciding whether to fork or not, and in the latter case to identify the particular result that should be reused:

```
import easyFJP.policy.MemoizationPolicy;
import easyFJP.policy.ExecutionContext;

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

The policy indicates EasyFJP to fork (and hence to ignore the contents of the cache) whenever the argument of a call to fib is an even number. Moreover, whenever shouldFork evaluates to false, EasyFJP attempts to reuse the value from the cache with the key as indicated by buildResultKey. However, if shouldFork evaluates to false but the key is invalid and leads to a cache miss, the normal execution (in parallel) takes place. Depending on the target execution environment for the application (e.g. multi-core, cluster), memoization works by using a local in-memory or a distributed cache. For example, our current distributed bindings rely on spymemcached [70], a general-purpose and very efficient distributed object caching system written in Java. We expect, however, to extend our implementation to support other caching technologies (see Section 4).

It is worth noting that we have explicitly implemented MyMemoizationPolicy so that the application does not reuse *every* previously computed result. This may not seem to be, in principle, a good tuning rule. However, memoization strategies like the one implemented by this policy, in which only a subset of already calculated results are reused, is useful in parallel optimization problems where creating a new fork for a subproblem may yield a better solution than reusing a similar, previously computed suboptimal result [2].

Moreover, for coding environment-specific policies, EasyFJP provides a well-defined interface to useful system metrics. To this end, EasyFJP provides two profiling modules: a local one, which is intended to be used when employing EasyFJP in conjunction with SMP parallel libraries, and a distributed one, which is useful when implementing policies for cluster and Grid applications. Within a single machine, metrics are gathered by using JMX [76], a platform-independent API for obtaining runtime information such as CPU load, available threads and memory, disk usage, etc. Moreover, to return cluster-wide values for the above metrics, EasyFJP implements a distributed monitoring service that predicts the global performance of both network and computational resources by using regression models and communicates these values via GMAC [40], a lightweight P2P protocol that provides efficient multicast services across distributed environments. With this support, users are able for example to query for the *overall* cluster CPU load or the amount of parallel runtime tasks under execution. In this light, by using the profiling API, a developer may code for instance a policy to relate the amount of parallelism of an application as an inverse function of the average CPU availability.

3.3. Step 3: Parallel code generation

Based on the synchronization points obtained from the source code analysis carried out at Step 1, the XML configuration of the input application, and the parallel library targeted by the developer, EasyFJP generates the final parallel application. To this end, for each class of the sequential application being parallelized, EasyFJP creates a *peer* class whose source code is derived from the sequential class but modified so that the peer exploits the target parallel library. Then, ordinary classes and created peers are seamlessly "wired" at load time by employing a simple bytecode rewriting technique. This essentially aims at avoiding modifying the source code of the original classes while supporting parallelism for them through those peers.

Basically, this technique takes advantage of the java.lang.instrument package, a built-in Java API that defines hooks for modifying classes at load time. The package is intended to be used through special

An Approach for Non-Intrusively Adding Malleable Fork/Join Parallelism into Ordinary JavaBeans Compliant Applications. (C. Mateos, A. Zunino, M. Campo). Computer Languages, Systems and Structures. Elsevier Science. ISSN: 1477-8424. Ed.: R. S. Ledley and B. Kalyanasundaram. Vol. 36, Num. 3, pp 288-315. 2010. DOI: doi:10.1016/j.cl.2009.12.003

libraries called Java agents, this is, pluggable user libraries –JAR (Java ARchive) files– that customize the class loading process. Java agents are accessed by the JVM every time an application requests to load a class. In our case, the classes that are subject to modification are the ones specified by the user as targets for parallelization in the corresponding XML configuration.

When rewriting a sequential class for such purposes, EasyFJP replaces the body of its divide and conquer method with a stub that delegates the execution of the method to the parallel counterpart in the associated peer. Therefore, for example, the fib method of the Fibonacci application of Section 3 is dynamically rewritten as:

```
public class Fibonacci implements Serializable{
  ...
  public long fib(long n){
    Fibonacci_Peer peer = new Fibonacci_Peer();
    copyProperties(this, peer);
    // Assuming we are targeting Satin
    ExecutorManager manager = ExecutorManagerFactory.getExecutor("Satin");
    return manager.execute(peer, "fib", new Object[]{n});
  }
  ...
}
```

, so that its computation is performed in parallel by the peer (Fibonacci_Peer), whose properties are instantiated via Java reflection from the running sequential object. Basically, copying properties is a generic procedure that is possible thanks to the uniformity provided by following the getters/setters convention of the JavaBean specification. Finally, ExecutorManager represents the EasyFJP class API that communicates with the middleware-level support that executes peers by performing the corresponding parallel library-specific initialization and disposal tasks. To generate peers, EasyFJP relies on builders, which are library-dependent components that know how to properly incorporate fork and join calls into sequential code. In addition, builders inject into peers the glue code to interact with the policies declared for the application. Let us illustrate all these transformations based on the Satin library, for which EasyFJP provides a builder.

To manually use Satin, divide and conquer methods considered for parallel execution must be identified through a *marker interface* that includes their exact signature and extends the *Spawnable* interface. The class containing parallel methods extends the *SatinObject* class and implements the marker interface. In addition, the invocations to parallel methods are stored in local variables. After specifying parallel methods and inserting synchronization calls into the application code, the developer must feed a compiled version of the application to the Satin compiler that translates, through Java bytecode instrumentation, each invocation to a parallel method into a Satin runtime task, so that a fork is issued at runtime. In a broad sense, bytecode instrumentation is the task of transforming the compiled version of a program to alter its semantics [28].

The purpose of the Satin builder is to automatically reproduce these tasks. The Satin builder generates the marker interface based on the operations specified within the XML configuration file of the application, and makes the peer extend and implement the required API classes and interfaces. Besides, the builder inserts appropriate calls to sync (the multi-fork join primitive of Satin) based on the output of the source code analysis of Step 1. Passing the source code of our example Fibonacci application through the builder (without taking into account policies) results in the generation of the following code:

```
1  // Marker interface
2  public interface Fibonacci_Marker extends satin.Spawnable{
3    public long fib(long n, long depth);
4  }
5  // Peer
6  public class Fibonacci_Peer extends satin.SatinObject
7      implements Fibonacci_Marker, Serializable{
8    // Properties are copied "as is" from the original class
9    ...
10 public long fib(long n){
11   return fib(n, 0);
12  }
13  // The Satin-enabled method, according to lines 2-4
14  public long fib(long n, int depth){
15    ...
```

An Approach for Non-Intrusively Adding Malleable Fork/Join Parallelism into Ordinary JavaBeans Compliant Applications. (C. Mateos, A. Zunino, M. Campo). Computer Languages, Systems and Structures. Elsevier Science. ISSN: 1477-8424. Ed.: R. S. Ledley and B. Kalyanasundaram. Vol. 36, Num. 3, pp 288-315. 2010. DOI: doi:10.1016/j.cl.2009.12.003

```
16      f1 = fib(n - 1, depth + 1);
17      f2 = fib(n - 2, depth + 1);
18      // The Satin multi-fork join primitive
19      super.sync();
20      return f1 + f2;
21    }
22    ...
23  }
```

A new method (lines 10-12) is added in order to invoke the actual parallel method (lines 14-21), whose code has been derived from the original fib method but modified to include Satin synchronization (line 19), and to keep track of the depth of the execution tree at runtime. This information, together with the current method parameters are encapsulated in an ExecutionContext object, which is used to feed policies by further modifying the source code of the newly generated parallel method. Figure 4 details the code transformations performed to support threshold policies (left) and memoization policies (right). Basically, before executing

```
public long fib(long n, int depth){
  ExecutionContext ctx =
     new ExecutionContext();
  ctx.setCurrentDepth(depth);
  ctx.addArgument(n);
  if (PolicyManager.fork(pId,ctx)){
    ...
    f1 = fib(n - 1, depth + 1);
    f2 = fib(n - 2, depth + 1);
    super.sync();
    return f1 + f2;
  }
  return fibSeq(n);
}
```

```
public long fib(long n, int depth){
  ExecutionContext ctx = new ExecutionContext();
  ctx.setCurrentDepth(depth);
  ctx.addArgument(n);
  if (!PolicyManager.fork(pId,ctx)){
    MemoizationPolicy mPolicy =
      PolicyManager.getMPolicy(pId);
    Object entry = CacheManager.get(
                   mPolicy.buildResultKey(ctx));
    if (entry != null)
      return (Long)entry;
  }
  ...
  f1 = fib(n - 1, depth + 1);
  f2 = fib(n - 2, depth + 1);
  super.sync();
  MemoizationPolicy mPolicy =
    PolicyManager.getMPolicy(pId);
  CacheManager.put(
    mPolicy.buildResultKey(ctx),f1+f2);
  return f1 + f2;
}
```

Figure 4: Source code transformations for injecting threshold policies (left) and memoization policies (right)

the target method in parallel, its associated policy (identified in the example as pId and extracted from the XML configuration of the application) is evaluated to test whether the execution of the method should be forked or executed sequentially instead. In this sense, peers keep an unmodified version of the original (sequential) divide and conquer methods, this is, fibSeq in our example (see Figure 4 (left)). From the point of view of source code generation, this mechanism shares many similarities with the two-clones strategy of the Cilk multithreaded language [36]. Roughly, based on explicit method-level directives for forking and coordinating methods, Cilk generates two versions (clones) of the method(s) being parallelized: a sequential one, which is executed when serial semantics are sufficient, and a parallel one, which contains calls to the Cilk API to exploit parallelism. However, from a development standpoint, EasyFJP does not rely on explicit code directives, but employs implicit parallelism and automatic techniques for spotting fork and joint points. Besides, EasyFJP is oriented towards generation of code for various target parallel libraries.

Furthermore, the code introduced to handle memoization policies uses a cache to store computed results. This latter behavior is transparently achieved by putting extra code right before each return sentence within a parallel method in order to update the contents of the cache. It is worth mentioning that EasyFJP also includes source code transformations to allow threshold and memoization techniques to simultaneously control the same parallel method. However, they have been omitted from the explanation for the sake of simplicity.

An Approach for Non-Intrusively Adding Malleable Fork/Join Parallelism into Ordinary JavaBeans Compliant Applications. (C. Mateos, A. Zunino, M. Campo). Computer Languages, Systems and Structures. Elsevier Science. ISSN: 1477-8424. Ed.: R. S. Ledley and B. Kalyanasundaram. Vol. 36, Num. 3, pp 288-315. 2010. DOI: doi:10.1016/j.cl.2009.12.003

3.4. EasyFJP: Applicability guidelines and related issues

Up to this point, we have described the parallelization and tuning process promoted by EasyFJP to parallelize sequential codes, which addresses the problems exhibited by current approaches to parallelism that usually assume expertise from developers in parallel technologies. However, determining whether a user code will effectively benefit from using EasyFJP depends on a number of issues that users should have in mind. This section explains what users should expect from our approach and what not.

One one hand, EasyFJP is designed to parallelize applications that follow the JavaBeans component specification –an extremely popular way to structure classes among Java developers– and obey some simple coding conventions. This positively impacts on the applicability of EasyFJP, and do not affect programmability, as the code structure expected by EasyFJP can be obtained by using the refactoring tools of modern Java IDEs. However, it is worth pointing out that 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.

Regarding the first issue, the choice of parallelizing an application (or an individual component operation) depends on whether the operation is suitable for being executed in parallel. 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 user. 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, this is, whether an execution environment running the desired parallel library is available or not. For example, a novice user would likely target a parallel library he knows is installed on a particular hardware, rather than the other way around.

On the other hand, 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, a subset of FJP applications can exploit caching techniques.

An issue that may affect the applicability of EasyFJP 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 [57], 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.

Finally, EasyFJP applications do not differ from the pack in terms of debuggability, in which parallel programming has been historically conceived as a notoriously hard task [65]. Specifically, 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. On the other hand, 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.

4. Prototype implementation

We have developed a proof-of-concept implementation of the EasyFJP approach to materialize the three steps described in the previous section. Our tool performs the initial code analysis over an in-memory tree structure derived from parsing an XML version of the sequential user application obtained through java2xml [48], a library for converting Java source to XML and viceversa. We are nevertheless working on porting the analyzer to Eclipse by implementing a plug-in. Eclipse provides an API to manipulate the abstract syntax tree of Java classes at a very deep level of detail. Targeting Eclipse will make our tool more attractive to Java developers and therefore will facilitate its adoption.

An Approach for Non-Intrusively Adding Malleable Fork/Join Parallelism into Ordinary JavaBeans Compliant Applications. (C. Mateos, A. Zunino, M. Campo). Computer Languages, Systems and Structures. Elsevier Science. ISSN: 1477-8424. Ed.: R. S. Ledley and B. Kalyanasundaram. Vol. 36, Num. 3, pp 288-315. 2010. DOI: doi:10.1016/j.cl.2009.12.003

As EasyFJP employs the agent support of Java to connect ordinary classes to their generated parallel counterpart (i.e. peers), the developer must enable the -javaagent JVM flag upon executing the application. This is, the startup command to execute the main application class must include -javaagent:easyFJP-agent.jar=<config-file>, where easyFJP-agent.jar is the implementation of the EasyFJP agent that rewrites sequential classes based on the XML configuration specified in config-file. Within the agent, class transformations are performed by using Javassist [22], a high-level library for dynamically instrumenting bytecodes.

As explained earlier, EasyFJP currently provides parallel code generation capabilities that target the Satin platform, which offers efficient scheduling algorithms to parallelize ordinary divide and conquer applications on local and wide-area clusters. However, we are adapting our ideas to target Doug Lea's framework, an API for programming concurrent SMP applications, and GridGain, a platform for developing master-worker applications on LANs and WANs. Basically, the former exposes an API to parallelize applications hiding many low-level details related to parallelism such as thread creation and synchronization from the programmer. Specifically, the API contains classes to manage thread pools, fork subtasks and wait for asynchronous results by exploiting the future object synchronization pattern. For example, the Doug Lea's framework version of the Fibonacci application would be:

```
1 import java.util.concurrent.Callable;
2 import java.util.concurrent.ExecutorService;
3 import java.util.concurrent.Future;
4 public class Fibonacci implements Callable<Long>{
5   private long n;
6   private ExecutorService pool;
7   public Fibonacci(long n, ExecutorService pool){
8     this.n = n;
9     this.pool = pool;
10  }
11  public Long call(){
12    return fib(this.n);
13  }
14  public Long fib(long n){
15    if (n < 2)
16      return new Long(n);
17    Future<Long> f1 = pool.submit(new Fibonacci(n - 1, pool));
18    Future<Long> f2 = pool.submit(new Fibonacci(n - 2, pool));
19    return f1.get() + f2.get();
20  }
21 }
```

The example employs an instance of the ExecutorService API class representing a pool of threads (lines 6-7) to asynchronously execute the recursive calls to fib in parallel. Since the pool internally maintains several threads, this code would automatically exploit SMP machines. Note that this framework implements an approach to synchronization based on a single-fork join primitive. In the above example, the points in which the application must wait for the parallel results of forked tasks are the accesses to the value of f1 and f2 at line 19. Furthermore, the GridGain platform builds upon this API, but supports the execution of such forked tasks on clusters. GridGain is also able to exploit SMP machines. In this sense, the two builders for these backends rely on the same generic heuristic for synchronization, which basically replaces each recursive call in the (sequential) input code by a call to the corresponding Callable object in the parallelized code, and also translates the places of the original application that access to subresults (in our case line 19) by using the associated future objects.

Regarding the EasyFJP policy framework, we have implemented the distributed caching support by using spymemcached [70], a Java client for memcached [32], a popular high-performance and distributed object caching backend. Nevertheless, we have designed the prototype of EasyFJP so that other distributed caching systems can be easily plugged. In this sense, alternatives include the Terracotta [80] object clustering platform, and a caching service on top of our GMAC P2P protocol [40], which is also used for implementing the monitoring service for collecting system metrics in EasyFJP.

An Approach for Non-Intrusively Adding Malleable Fork/Join Parallelism into Ordinary JavaBeans Compliant Applications. (C. Mateos, A. Zunino, M. Campo). Computer Languages, Systems and Structures. Elsevier Science. ISSN: 1477-8424. Ed.: R. S. Ledley and B. Kalyanasundaram. Vol. 36, Num. 3, pp 288-315. 2010. DOI: doi:10.1016/j.cl.2009.12.003

5. Experimental evaluation

In this section, we describe the experiments that were performed to evaluate the practical soundness of EasyFJP. We evaluated two essential aspects of our approach: we assessed the effectiveness of its generic heuristic for inserting synchronization barriers and we quantified the benefits and potential overheads regarding the policy support of EasyFJP. In addition, to provide a down-to-earth evaluation of EasyFJP, we also performed the parallelization of a real-world application. Basically, Sections 5.1 and 5.2 describe the evaluations of EasyFJP using small to moderate computational granularities, whereas Section 5.3 reports experiments with large task granularities.

The first evaluation was performed by running EasyFJP, targeting the Satin parallel library, and pure Satin versions of a number of CPU-intensive classic divide and conquer applications on a local cluster and a wide-area cluster. As we wanted to accurately test the effects of our multi-fork join insertion techniques in the performance of the test applications in these two environments, we temporarily disabled the injection of the code for supporting policies illustrated in Figure 4. These experiments are reported in Section 5.1.

In a second round of tests, we enabled the injection of policy code to quantify the incidence as well as the effectiveness of the EasyFJP policy support for two fork-intensive benchmark applications when using both threshold and memoization policies. These experiments are presented in Section 5.2, and were run on a local cluster to better evaluate the policy layer.

Finally, the third evaluation involved the parallelization via both EasyFJP and Satin of a sequential implementation of a sequence alignment application, a common problem in the area of bioinformatics. Roughly, sequence alignment is the process of comparing DNA or protein sequences to find similarities. This evaluation is reported in Section 5.3.

It is worth noting that it is out of the scope of this paper to evaluate the performance of the distributed monitoring service for environment-specific policies. A rigorous evaluation of this support in terms of both the effectiveness of this kind of policies and the efficiency of its underlying GMAC protocol can be found in [64] and [40], respectively.

5.1. Evaluation of the heuristic for inserting synchronization

This evaluation involved the execution of five CPU-intensive applications, which for the sake of fairness were obtained from the Satin project [84]. To obtain the EasyFJP versions of these applications, we first removed from their source code any sentence related to parallelism and/or application tuning, to derive the sequential divide and conquer counterparts of the applications. Then, we used the EasyFJP binding to Satin to obtain the parallel implementations of the sequential codes, but without including API code for supporting policies. Table 2 summarizes the applications and the parameters used in the experiments.

To run the tests, we used a cluster of 15 machines running Mandriva Linux 2009.0, Java 5 and Satin 2.1 connected through a 100 Mbps LAN. We used 8 single core machines with a 2.80 MHz CPU and 1.25 MB of RAM, and 7 single core machines with a 3 MHz CPU and 1.5 MB of RAM. In spite of that, as discussed in Section 4, EasyFJP targets multi-core parallel libraries, we used single-core machines in the experiments since Satin is not designed to directly exploit multi-core machines. Figure 5 shows the average execution time for 25 runs of these applications. In all cases, deviations were below 5%. Despite being an acceptable deviation when experimenting on wide area Grids, this percentage is rather high for a LAN-based cluster. The cause of this effect is that Satin –and thus the EasyFJP and pure Satin applications– relies on a random task stealing algorithm.

EasyFJP performed in a very competitive way compared to Satin, despite the fact that EasyFJP employs an heuristic algorithm for inserting synchronization and introduces some technological noise, which intuitively should translate into performance overhead. Basically, this noise is caused by the Java agent that wires ordinary application and peers together and the library-dependent executor objects that handle the execution of parallel methods. For 3 out of 5 test applications (PF, Cov, MM), EasyFJP introduced performance gains with respect to Satin, which could be explained in part by the random nature of the Satin scheduler, the differences between the number of calls to sync and the places of the application code in which these calls are located. Naturally, these differences stem from the fact that the Satin versions of the applications were parallelized and therefore provided with synchronization by hand, while the EasyFJP counterparts were parallelized by applying our heuristic on the sequential recursive codes derived from the pure Satin applications. However, our goal is not to outperform existing parallel libraries, but simplifying

An Approach for Non-Intrusively Adding Malleable Fork/Join Parallelism into Ordinary JavaBeans Compliant Applications. (C. Mateos, A. Zunino, M. Campo). Computer Languages, Systems and Structures. Elsevier Science. ISSN: 1477-8424. Ed.: R. S. Ledley and B. Kalyanasundaram. Vol. 36, Num. 3, pp 288-315. 2010. DOI: doi:10.1016/j.cl.2009.12.003

Table 2: Test applications used for evaluating the heuristic for identifying multi-fork join points

Application	**Description**	**Run size**
Prime factorization (PF)	Splits an integer I into its prime factors. The multiplication of these factors is equal to I	$I = 15,576,890,767$
The set covering problem (Cov)	Finds a minimal number of subsets from a list of sets L which covers all elements within L. The problem takes as a parameter the size of L	List of 33 sets with random elements
The knapsack problem (KS)	Finds a set of items, each with a weight W and a value V, so that the total value is maximal, while not exceeding a fixed weight limit. The problem receives as a parameter the initial number of items	A list of 32 items with random weights and values
Matrix multiplication (MM)	Implements the popular Strassen's algorithm	A matrix of $3,072x3,072$ with random cell values
Adaptive numerical integration (Ad)	Approximates a function $f(x)$ within a given interval (a,b) by replacing its curve by a straight line from $(a, f(a))$ to $(b, f(b))$. The application receives as parameters $f(x)$, a, b, and an $epsilon$ that controls the mechanics of the algorithm	$f(x) = 0.1 * x * sin(x)$, $a = 0$, $b = 250,000$, $epsilon = 0.000001$

their usage without incurring in an excessive penalty in terms of performance. This experiment shows that EasyFJP facilitates the construction of Satin applications, while stays competitive compared to directly employing Satin, which is explained by the effectiveness of our generic heuristic.

Later, we executed a subset of the above applications on a wide-area cluster, which was established by using WANem [79], a software for simulating WAN conditions over a LAN. We simulated 3 Internet--connected local clusters C_1, C_2 and C_3 by using 4, 5 and 6 of the machines of the local cluster, respectively. Each WAN link was a T1 connection with a round-trip latency of 200 ms and a jitter of 10 ms. Both the EasyFJP and the pure Satin variants of the applications were configured to use the Cluster-aware Random Stealing (CRS) [84] algorithm of the Satin framework, instead of its default Random Stealing algorithm. With CRS, when a machine becomes idle, it attempts to steal an unfinished task from both remote or local machines, but intra-cluster steals have a greater priority than wide-area steals, saving bandwidth and minimizing WAN latencies. Furthermore, the computation to network data transfer ratio of *MM* in this setting was very small, which severely and negatively affected processor usage. Therefore, we decided to left the application out of the analysis since it did not experienced a CPU-bound behavior in this testbed. Table 3 summarizes the obtained results for both the local and the wide-area clusters.

Figure 6 depicts the average execution time for 40 runs of the selected applications. In all cases, deviations were around 11%, which as explained before is mainly due to the random nature of the Satin scheduler plus the fact that we used WAN links with jitter to connect the local clusters. As shown, the EasyFJP applications performed better than their respective Satin versions. For the case of *KS* and *Ad*, and unlike the previous LAN experiment, EasyFJP outperformed Satin. Moreover, for the case of *PF* and *Cov*, the performance gains introduced by EasyFJP were even greater than the gains obtained for these applications in the LAN setting (1-2% versus 5-6%). Hence, it seems that executing these four applications in the wide-area cluster accentuated the differences regarding the way synchronization is added when using both tools. However, this trend should be further corroborated. All in all, these results show that the EasyFJP applications performed very well, which aligns with the promissory results obtained in the LAN environment.

As a complement, Figure 7 depicts the speedup factor achieved by the Satin and EasyFJP variants when executing the test applications in the local-area cluster (left) and the wide-area cluster (right). This

An Approach for Non-Intrusively Adding Malleable Fork/Join Parallelism into Ordinary JavaBeans Compliant Applications. (C. Mateos, A. Zunino, M. Campo). Computer Languages, Systems and Structures. Elsevier Science. ISSN: 1477-8424. Ed.: R. S. Ledley and B. Kalyanasundaram. Vol. 36, Num. 3, pp 288-315. 2010. DOI: doi:10.1016/j.cl.2009.12.003

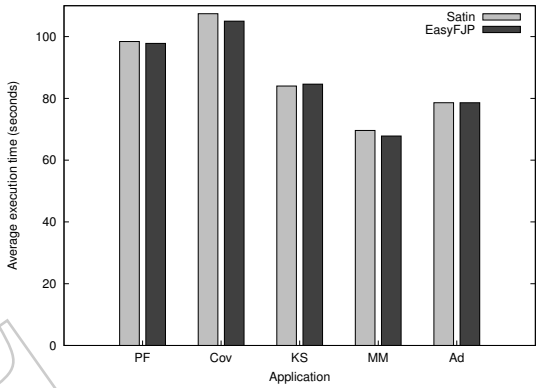


Figure 5: Performance of the test applications in the local cluster

Table 3: Evaluation of the heuristic for inserting synchronization: Performance results

Application	**Satin**		**EasyFJP**	
	Average execution time (seconds)		**Average execution time (seconds)**	
	Local cluster	**Wide-area cluster**	**Local cluster**	**Wide-area cluster**
Prime factorization (PF)	98.4	129.0	97.8	122.4
The set covering problem (Cov)	107.4	138.0	105.0	129.6
The knapsack problem (KS)	84.0	117.0	84.6	111.6
Matrix multiplication (MM)	69.6	N/A	67.8	N/A
Adaptive numerical integration (Ad)	78.6	117.0	78.6	102.6

factor was computed as T_s/T_p, where T_s and T_p are the times required to execute the sequential and parallel versions of these applications, respectively. Basically, to compute T_s, the sequential codes where run on the machine of the experimental setting with the best hardware capabilities in terms of CPU and memory. The figures also depict the theoretical maximum speedup factor, given by the number of available machines in either experimental settings, this is, fifteen. Moreover, note that all the applications executed in the local-area cluster achieved important speedups except for *MM*, which is due to its low computation to data transfer ratio with respect to the rest of the applications. Overall, the implications of the obtained speedups are twofold. First, the original applications certainly benefited from being parallelized, which makes them representative to provide the basis for a significant evaluation of our synchronization heuristics. Second, EasyFJP achieved speedups levels that are competitive to those achieved by Satin in both settings.

These positive results are consistent with the main goal of the synchronization techniques of EasyFJP, which is to automatically incorporate synchronization as similar as a human programmer would do. It is worth emphasizing that we cannot conclude from the tests that our heuristic is better than that of Satin. Specifically, despite the fact that EasyFJP outperformed Satin in the WAN testbed for the above benchmarks, Satin and EasyFJP performance were similar for the sequence alignment application of Section 5.3 in the same setting. In this sense, results suggest that even when not fully exploiting policies, applications parallelized via our automatic synchronization techniques are *competitive* in terms of performance compared to the manual approach to parallelism and synchronization followed by Satin.

An Approach for Non-Intrusively Adding Malleable Fork/Join Parallelism into Ordinary JavaBeans Compliant Applications. (C. Mateos, A. Zunino, M. Campo). Computer Languages, Systems and Structures. Elsevier Science. ISSN: 1477-8424. Ed.: R. S. Ledley and B. Kalyanasundaram. Vol. 36, Num. 3, pp 288-315. 2010. DOI: doi:10.1016/j.cl.2009.12.003

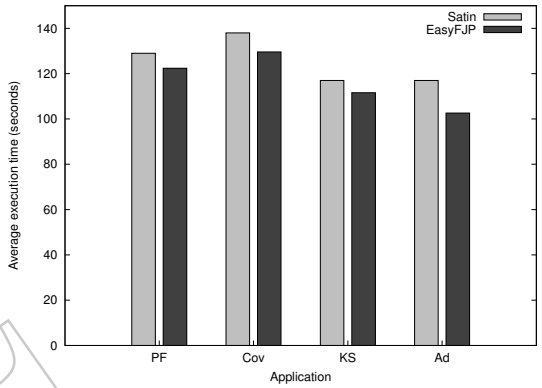
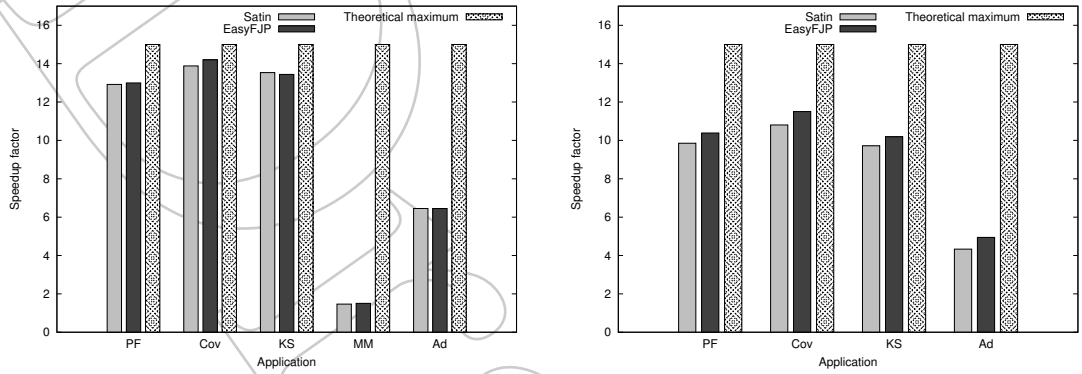


Figure 6: Performance of the test applications in the wide-area cluster



(a) Local-area cluster

(b) Wide-area cluster

Figure 7: Speedup

5.2. Evaluation of the policy support

The second part of the evaluation involved the execution of divide and conquer versions of two benchmark applications, namely the *Nth* Fibonacci and the binomial coefficient (also known as "N over K"). Similarly to the experiments reported in the previous subsection, we processed the source codes with EasyFJP by enabling the injection of code for supporting policies, and we coded alternative parallel versions with Satin. Then, we derived variants of the EasyFJP and Satin implementations by introducing threshold and memoization. Roughly, the purpose of the evaluation was to assess the overhead and effectiveness of tuning applications through policies versus using the same optimization rules within the application code, which was the case of the Satin versions. For the Fibonacci application we used $N = 42$, whereas for the binomial coefficient application we used $N = 33$ (K was set to $N/2$). Note that with these arguments, the number of tasks generated by the applications at runtime was huge, thus it was a rather challenging scenario to EasyFJP.

First, we obtained the Satin versions of the Fibonacci and N over K applications by manually parallelizing their sequential recursive codes, and then we derived four variants of the parallelized codes by including threshold optimizations so as to compute in parallel the subtasks associated to any input N provided N is greater or equal to some given threshold. Furthermore, we obtained two more variants by using a simple memoization strategy that stored the result of a subtask provided the associated N is below some limit (for consistency purposes, we employed similar limits to the threshold-based variants). Memoization in Satin was implemented by means of *shared objects* [83], a mechanism provided by Satin to transparently share and update the state of a Java object (in this case a cache) among the distributed parallel computations of an executing application. These variants were carefully designed and implemented to perform as few write

An Approach for Non-Intrusively Adding Malleable Fork/Join Parallelism into Ordinary JavaBeans Compliant Applications. (C. Mateos, A. Zunino, M. Campo). Computer Languages, Systems and Structures. Elsevier Science. ISSN: 1477-8424. Ed.: R. S. Ledley and B. Kalyanasundaram. Vol. 36, Num. 3, pp 288-315. 2010. DOI: doi:10.1016/j.cl.2009.12.003

operations on the shared object as possible, which are the operations broadcasted by Satin and thus the ones that generate network traffic.

Similarly, based on the sequential Fibonacci and N over K applications, we generated the corresponding EasyFJP variants. Basically, the unoptimized variants used a policy that always forks subtasks, while the rest of the variants implemented the above optimization rules through the use of threshold and memoization policies. Interestingly, this only involved configuring different policies for the two template parallel applications that were first obtained via the EasyFJP code generator. Table 4 shows the number of source code lines of the policy-based test applications after parallelizing the original sequential codes via both Satin and EasyFJP. This can be used as a very rough estimation of the tuning effort in either cases. In order to carry out a fair comparison, before taking metrics, the codes were transformed to a uniform formatting standard, and Java import statements were optimized by employing the source code optimizing tools of the Eclipse IDE. As shown in the Table, the unoptimized EasyFJP variants had more source code lines than their corresponding unoptimized Satin variants because we enabled back the generation of instructions to support policies. However, these instructions are injected into sequential codes automatically. The Table also includes the code size of the optimized variants. The threshold-based EasyFJP variants had some more lines than their Satin counterparts, but memoization required less source code in EasyFJP, since most of the behavior for caching results is transparently managed at the framework level.

Table 4: Policy-based test applications: source code lines

Application	**Original (sequential)**	**Satin**			**EasyFJP**		
		Unoptimized	**Threshold**	**Memoization**	**Unoptimized**	**Threshold**	**Memoization**
Fibonacci	29	40	57	73	54	66	65
N over K	32	49	66	115	70	81	79

Table 5 shows the average execution time (in seconds) for 25 executions of the applications on the local cluster described in the previous subsection under the same execution conditions. For each variant (leftmost column), we considered three different executables by varying the granularity of the runtime tasks: *extra fine* (i.e. the unoptimized/optimized parallel codes), *fine* and *medium*. These two latter were obtained by adding to these codes a fake counting loop up to two different numbers within the body of the recursive methods. For the Satin and EasyFJP variants using medium granularity and thresholds, the obtained execution times were slightly better or in some cases even worse than their associated unoptimized variants. Nevertheless, for the sake of consistency, the threshold values used in those experiments were the same as the extra fine-grained and fine-grained variants.

In addition, the Table shows the amount of runtime tasks generated by each variant of the test applications. For the case of the unoptimized applications and the variants using thresholds, the values were independent of the tool and the execution. However, for the variants using memoization, the values varied between the two tools and even between different executions employing the same tool, because under these variants the amount of subtasks is subject to dynamic factors as a consequence of relying on two approaches to distributed object sharing, this is, an object replication mechanism for the case of Satin shared objects, and a spoke-hub object distribution scheme for the case of EasyFJP/memcached.

With respect to the resulting execution times, it can be observed from the Table that EasyFJP incurred in some overheads compared to Satin for the executables with extra fine granularity. Basically, this is due to the extremely small granularity (CPU requirements) of their parallel tasks, which in turn caused the EasyFJP executables to spend more time asking whether to fork or not, than doing useful computations. In other words, in this scenario, evaluating policies is more expensive than executing the associated subtasks itself. Contrarily, for the Satin variants, deciding whether to fork or not was cheaper, since this behavior was absent in the unoptimized applications and hardcoded in the code of their optimized counterparts. In this line, the amount of injected parallel tasks to the cluster per time unit was smaller for EasyFJP, which caused cluster nodes to have less tasks available to execute at any given time, as evidenced by the low percentage of task steals (i.e. amount of successful task steals over the amount of steal attempts).

An Approach for Non-Intrusively Adding Malleable Fork/Join Parallelism into Ordinary JavaBeans Compliant Applications. (C. Mateos, A. Zunino, M. Campo). Computer Languages, Systems and Structures. Elsevier Science. ISSN: 1477-8424. Ed.: R. S. Ledley and B. Kalyanasundaram. Vol. 36, Num. 3, pp 288-315. 2010. DOI: doi:10.1016/j.cl.2009.12.003

Table 5: Evaluation of the EasyFJP policy mechanism: Performance results

Application	**# of subtasks**	**Satin**			**EasyFJP**		
		Average execution time (seconds)			**Average execution time (seconds)**		
		Extra fine	**Fine**	**Medium**	**Extra fine**	**Fine**	**Medium**
Fibonacci (unoptimized)	866,988,873	62.69	102.41	284.84	76.97	118.52	303.05
Fibonacci (threshold=4)	331,160,281	24.60	71.19	276.40	30.01	76.31	280.48
Fibonacci (threshold=5)	204,668,309	15.52	66.40	288.86	19.49	70.75	292.97
Fibonacci (memoization)	Avg. Satin (204,668,383); Avg. EasyFJP (204,668,315)	30.03	39.32	82.90	25.32	34.45	34.16
N over K (unoptimized)	1,131,445,439	82.70	138.94	392.26	100.47	155.86	400.72
N over K (threshold=5)	549,754,738	40.26	108.23	400.89	50.25	117.98	410.27
N over K (threshold=6)	333,793,708	25.27	95.46	402.04	31.51	102.41	408.51
N over K (memoization)	Avg. Satin (530,365,082); Avg. EasyFJP (530,365,051)	71.57	96.43	211.17	59.92	84.07	83.40

However, the tests with the extra fine-grained variants of the applications served us as a basis for further comparison, since FJP clearly benefits problems which can be split into several CPU-intensive but less granular tasks. In this sense, Figure 8 shows the performance overhead (percentage) of the EasyFJP applications with respect to their Satin counterparts as the granularity of the runtime tasks slightly increases. The Figure shows the results for the unoptimized variants and the variants using threshold policies. It can be seen that the average percentage overhead decreased dramatically, as the performance penalty introduced by the policy framework rapidly became small with respect to the total execution time of the applications. Besides, the overheads just involve performance penalties in the order of seconds for applications that take few minutes to execute. Certainly, for typical CPU-intensive parallel applications, which usually comprise a number of coarse-grained tasks that together take several minutes or even hours to finish, the overheads would intuitively be insignificant. This reasonable extrapolation suggests that developers can take advantage of the non-invasive nature of the policy mechanism and at the same time delivering competitive performance for typical FJP applications.

Figure 9 shows the speedups introduced by the optimized variants (i.e. the ones using policies) of the test applications with respect to their unoptimized parallel variants, computed according to the formula T_u/T_{pol}, being T_u and T_{pol} the time required to run the unoptimized and policy-based variants of the Fibonacci and N over K applications, respectively. On one hand, for the variants using threshold-based optimizations, the effectiveness of EasyFJP policies proved to be competitive (see Figure 9 (a) and Figure 9 (b)). As explained before, some of the codes with medium granularity did not improve from employing threshold rules (i.e. the bars with speedup factor < 1), because as the task granularity increases, the performance gains that may result from avoiding forking tasks decrease. Nevertheless, the obtained results show that the effectiveness levels achieved by EasyFJP threshold policies were similar to those achieved by Satin. On the other hand, the EasyFJP variants based on memoization outperformed their respective Satin counterparts (see Figure 9 (c) and Figure 9 (d)). Memoization in Satin was achieved through a shared object, a built-in Satin mechanism by which parallel tasks were supplied with an up-to-date local copy of

An Approach for Non-Intrusively Adding Malleable Fork/Join Parallelism into Ordinary JavaBeans Compliant Applications. (C. Mateos, A. Zunino, M. Campo). Computer Languages, Systems and Structures. Elsevier Science. ISSN: 1477-8424. Ed.: R. S. Ledley and B. Kalyanasundaram. Vol. 36, Num. 3, pp 288-315. 2010. DOI: doi:10.1016/j.cl.2009.12.003

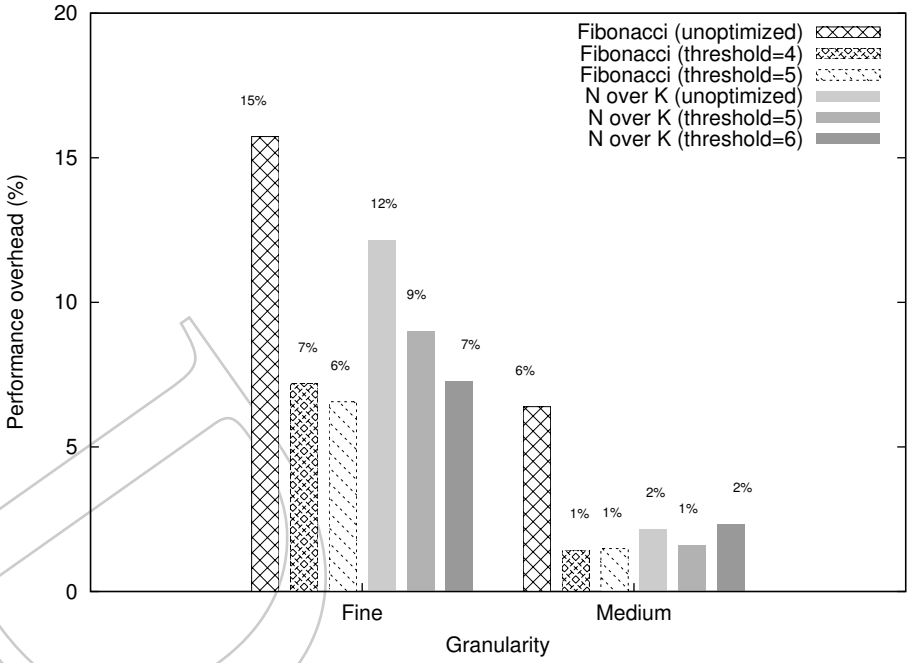
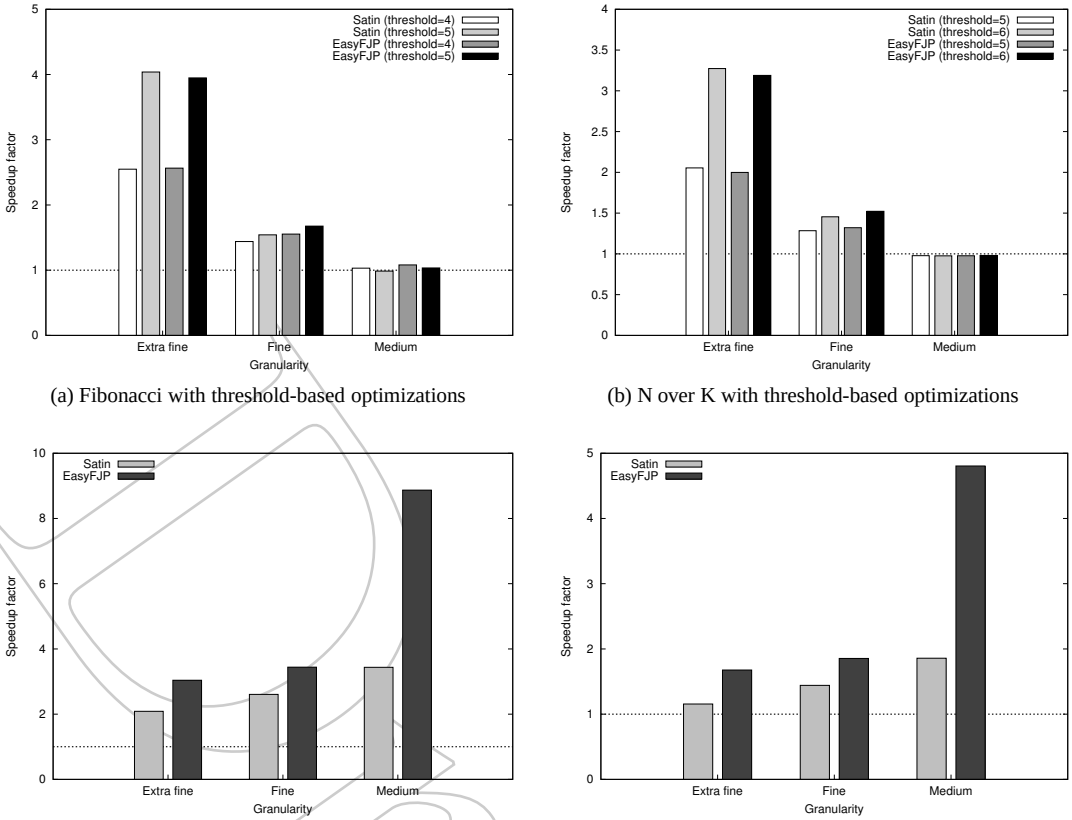


Figure 8: Performance overhead of the EasyFJP applications with respect to their Satin counterparts as the task granularity increases

the results cache. Upon creating and forking subtasks, its parent task must pass on to them a pointer to the shared object, which potentially involves moving through the network more state when subtasks are stolen by remote nodes for execution. Alternatively, our current caching scheme uses a spoke-hub architecture, by which cluster nodes maintain a local frontend (or level 1) cache in front of one or more backend (or level 2) cache servers. In this way, communication between local and remote caches are performed only when a miss occurs at a frontend cache. As depicted, in this experimental scenario, our caching support proved to be more efficient than Satin shared objects, for the three granularities. However, the semi-centralized nature of this support may lead to scalability issues when using larger clusters, for which Satin shared objects are designed. As mentioned in Section 4, we are working on scalable distributed caching support to address such problems.

Complementary, Figure 10 depicts the speedup factor T_s/T_{pol} of the variants using policies with respect to the sequential versions of the Fibonacci and N over K applications. Basically, for the variants based on threshold policies, the obtained speedups confirm the discussed trend of the behavior of the policies support illustrated in Figure 8, this is, the larger the granularity or computational size of the parallel tasks, the less the negative effect of the policy framework in the execution times of the policy-based applications. On the other hand, the EasyFJP applications using memoization policies significantly improved speedup as task granularity increased while outperformed their Satin counterparts. In any case, the better-than--ideal speedup factors obtained in some executions are explained by the well-known “super linear speedup” effect that arises when paralleling a sequential program along with an extensive use of caching techniques for avoiding recalculating subresults. The same effect is usually observed, for example, when performing backtracking in parallel and dynamically allowing the resulting program to dynamically prune branches of the initial exhaustive search space.

An Approach for Non-Intrusively Adding Malleable Fork/Join Parallelism into Ordinary JavaBeans Compliant Applications. (C. Mateos, A. Zunino, M. Campo). Computer Languages, Systems and Structures. Elsevier Science. ISSN: 1477-8424. Ed.: R. S. Ledley and B. Kalyanasundaram. Vol. 36, Num. 3, pp 288-315. 2010. DOI: doi:10.1016/j.cl.2009.12.003



(a) Fibonacci with threshold-based optimizations

(b) N over K with threshold-based optimizations

(c) Fibonacci with memoization-based optimizations

(d) N over K with memoization-based optimizations

Figure 9: Speedups of the optimized test applications with respect to their unoptimized parallel counterparts

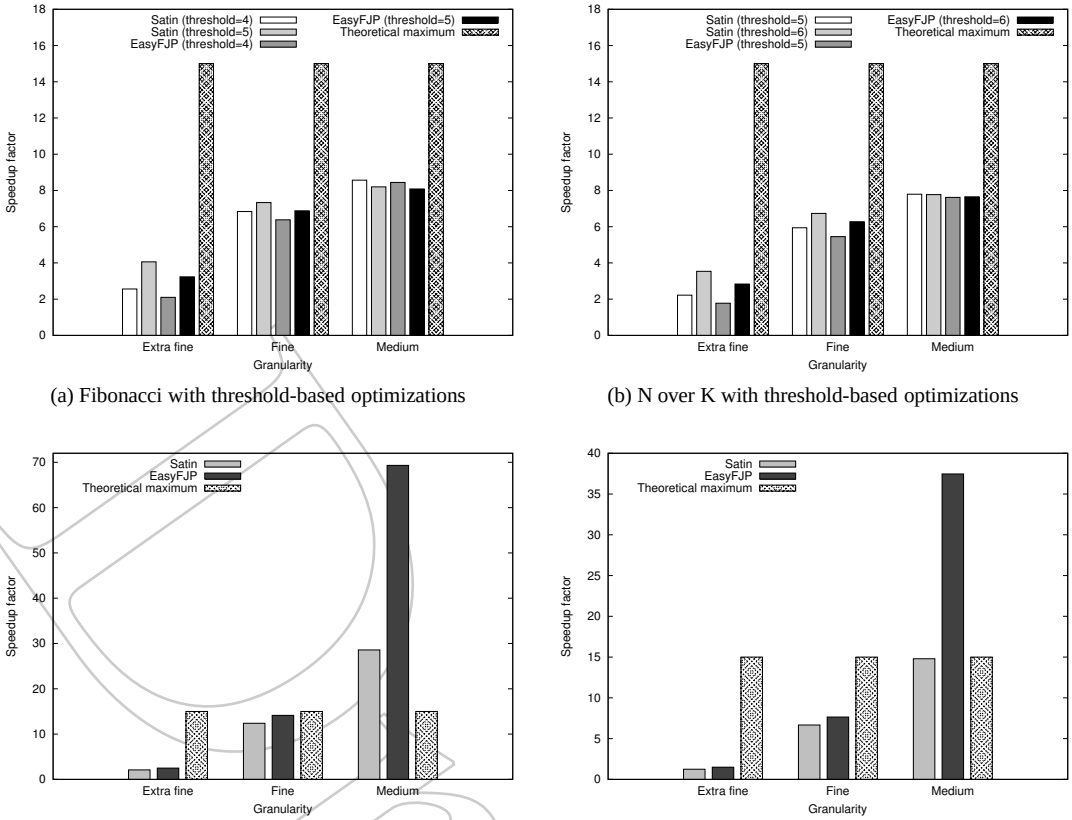
5.3. A real-world example: Sequence alignment

The third evaluation involved the execution of an application for local pairwise sequence alignment[3]. Broadly, this application represents a biological entity such as a gene in a computer-interpretable way (e.g. strings of characters) and manipulates the resulting representation by using sequence alignment algorithms. We first obtained a sequential fork-join implementation of this application by adapting an existing sequence alignment code extracted from the JPPF project [74], and then we parallelized it by using Satin and EasyFJP. The original application used the jaligner [68] library, an open source implementation of an improved version of the Smith-Waterman algorithm [39]. Given a pair of sequences, the algorithm outputs a coefficient that represents the level of similarity between these two by using a scoring matrix from a set of predefined matrixes. To execute the experiments, we used the PAM120 matrix, which works well in most cases.

The application aligned an unknown input target sequence against an entire sequence database, which was replicated across the nodes of our experimental wide-area Grid to allow local access to the sequence data. The application operated by dividing the portions of the data to compare against into two different subproblems until a certain threshold on the data was reached. We used the same thresholds for both Satin and EasyFJP. Moreover, we compared input sequences against real-world databases of Influenza A sequences extracted from the National Center for Biotechnology Information (NCBI) Web site[4]. The NCBI is an organization devoted to computational biology that maintains public genomes databases, disseminates biomedical information and develops bioinformatics software. Concretely, we used the databases shown

[3] http://en.wikipedia.org/wiki/Sequence_alignment
[4] http://www.ncbi.nlm.nih.gov

An Approach for Non-Intrusively Adding Malleable Fork/Join Parallelism into Ordinary JavaBeans Compliant Applications. (C. Mateos, A. Zunino, M. Campo). Computer Languages, Systems and Structures. Elsevier Science. ISSN: 1477-8424. Ed.: R. S. Ledley and B. Kalyanasundaram. Vol. 36, Num. 3, pp 288-315. 2010. DOI: doi:10.1016/j.cl.2009.12.003



(a) Fibonacci with threshold-based optimizations

(b) N over K with threshold-based optimizations

(c) Fibonacci with memoization-based optimizations

(d) N over K with memoization-based optimizations

Figure 10: Speedups of the optimized test applications with respect to their pure sequential counterpart

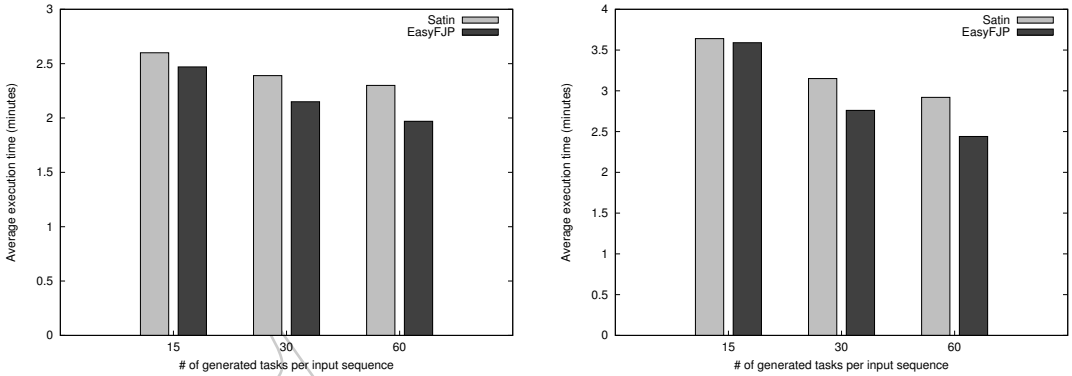
in Table 6. It is worth noting that the tests conceived the EasyFJP implementation of the application as a mean to provide more evidence about the performance of EasyFJP, but it is not our goal to come out with a better implementation of sequence alignment in Grid settings, for which specialized frameworks such as mpiBLAST [9] and G-BLAST [85] already exist.

Database	Size (# of sequences)	Size (MB)	Host	Period
DB 1	9,620	4.8	Human	Jan-2007/Dec-2008
DB 2	12,325	6.2	Human	Jan-2006/Dec-2008
DB 3	19,745	7.5	Human	Jan-2004/Dec-2008
DB 4	42,334	21.4	Avian	All registered cases up to now

Table 6: Protein sequence databases used in the experiments

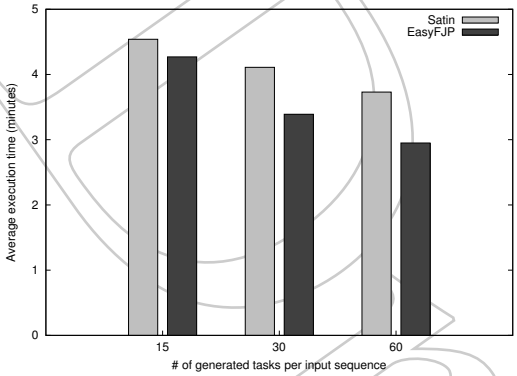
As mentioned earlier, the objective of this evaluation was to assess the effects of the parallelization mechanisms of EasyFJP in the performance of a complex application while using more coarse task granularities than the ones employed in the experiments reported in Sections 5.1 and 5.2. To this end, we derived several variants of each implementation of the sequence alignment application by varying the computational granularity of the resulting runtime tasks, this is, we assigned large portions of the database to each forked task for aligning purposes. This allowed us to better evaluate the effect of our heuristic for inserting

An Approach for Non-Intrusively Adding Malleable Fork/Join Parallelism into Ordinary JavaBeans Compliant Applications. (C. Mateos, A. Zunino, M. Campo). Computer Languages, Systems and Structures. Elsevier Science. ISSN: 1477-8424. Ed.: R. S. Ledley and B. Kalyanasundaram. Vol. 36, Num. 3, pp 288-315. 2010. DOI: doi:10.1016/j.cl.2009.12.003

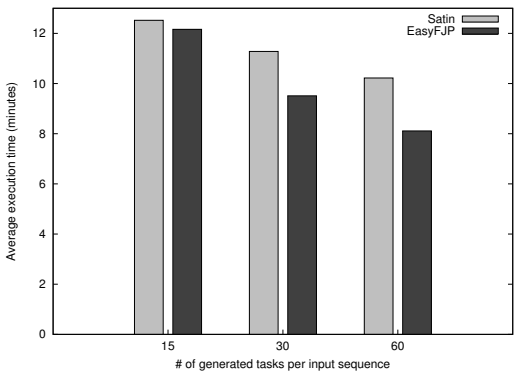


(a) DB1 (9,620 sequences)

(b) DB2 (12,325 sequences)



(c) DB3 (19,745 sequences)



(d) DB4 (42,334 sequences)

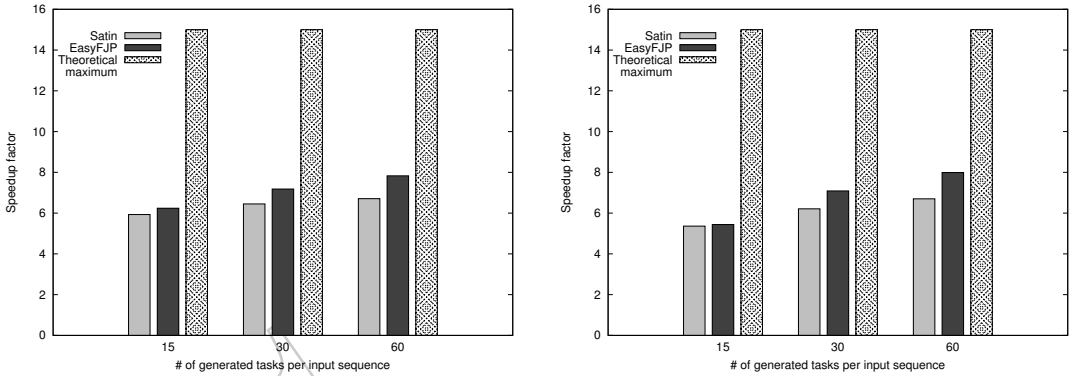
Figure 11: Performance of the sequence alignment application

join points in the resulting performance since not fully exploiting policies clearly reduces the incidence of this latter mechanism in the execution times. In other words, in opposition to the evaluation in Section 5.2, using coarse granularities means that the policies configured to the EasyFJP applications are invoked only few times during their execution and thus the effect of the policy framework is dramatically reduced. Finally, parallelizing code extracted from a neutral implementation of the sequence alignment application (i.e. based on JPPF) allowed us to also consider in the tests the implications of both Satin and EasyFJP on the input application. In this way, the process of obtaining the different parallel variants from the same initial source code in either cases conveys the framework-specific mechanisms for controlling the synchronization and the granularity of the resulting parallel tasks at runtime. With respect to source code lines, the original JPPF application had 327 lines, whereas the Satin and EasyFJP counterparts had similar number of lines, this is, around 440.

Figure 11 shows the average execution time for 40 runs of the Satin and EasyFJP variants of the application. For each target database, we run three different versions of these two alternatives by varying the computational granularity, or in this case, the size of the portion of the database that is analyzed per forked task. Again, the EasyFJP variants were implemented via threshold policies, whereas the Satin ones controlled such granularity by including thresholds directly into the application code. On one hand, we considered a very coarse granularity, by which the application was instructed to generate one task per available cluster node (i.e. 15 tasks in total to align a target sequence), plus somewhat optimized, more granular variants that injected 30 and 60 tasks into the Grid at runtime. Furthermore, as the sequence alignment application is not only CPU-intensive but also data-intensive, we did not achieved a very significant CPU load when aligning one target sequence per execution. Therefore, we decided to process two input sequences simultaneously per execution.

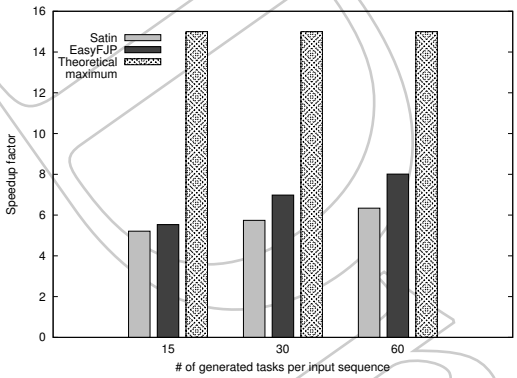
As illustrated in the figure, EasyFJP behaved better than Satin for all databases. Deviations were con-

An Approach for Non-Intrusively Adding Malleable Fork/Join Parallelism into Ordinary JavaBeans Compliant Applications. (C. Mateos, A. Zunino, M. Campo). Computer Languages, Systems and Structures. Elsevier Science. ISSN: 1477-8424. Ed.: R. S. Ledley and B. Kalyanasundaram. Vol. 36, Num. 3, pp 288-315. 2010. DOI: doi:10.1016/j.cl.2009.12.003

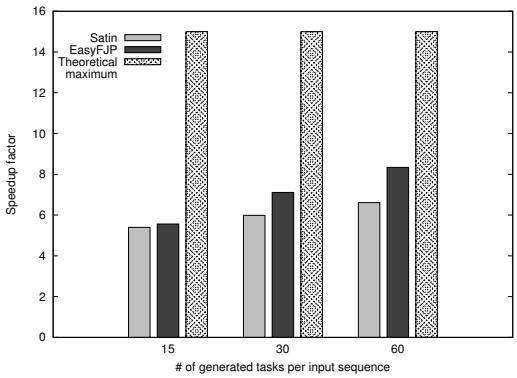


(a) DB1 (9,620 sequences)

(b) DB2 (12,325 sequences)



(c) DB3 (19,745 sequences)



(d) DB4 (42,334 sequences)

Figure 12: Speedups of the sequence alignment application with respect to their pure sequential counterpart

sistently in the range of 2-5% for the case of the EasyFJP variants, whereas they were between 2-11% for the case of Satin with slightly greater deviations as the number of forked tasks increased. The most interesting aspect of these results are, on one hand, that the heuristic for inserting synchronization of EasyFJP, evaluated in the tests via the variants with maximum computational granularity, leads to competitive performance not only for the benchmark applications discussed before but also for complex applications. On the other hand, the EasyFJP variants of the application relying on the other two granularities (i.e. the ones creating 30 and 60 tasks per input sequence) outperformed their Satin counterparts, which shows that the policy framework is useful for real-world applications. As a corollary, this latter implication is consistent with the results reported in Section 5.2, which suggested that the administrative overhead of policies may be negligible for parallelized applications relying on coarse task computational granularities. Finally, Figure 12 depicts the speedup factor achieved by the parallel sequence alignment applications with respect to the sequential version, which was executed on the fastest cluster machine. Interestingly, for the variants generating 30 and 60 tasks, EasyFJP improved the speedups of Satin by a factor of up to 1.24 and 1.73, respectively.

6. Conclusions and future work

In this article, we presented EasyFJP, an approach to semi-automatically and non-invasively introducing FJP into divide and conquer sequential applications. The main goal of EasyFJP is to isolate application logic as much as possible from the code that performs asynchronous task execution, synchronization and tuning of applications. The benefits of this approach are twofold. On one hand, users who are not proficient in parallel programming are allowed to quickly obtain parallel counterparts of their sequential applications, which in the short term may help “sequential” Java developers to gradually move into parallel program-

An Approach for Non-Intrusively Adding Malleable Fork/Join Parallelism into Ordinary JavaBeans Compliant Applications. (C. Mateos, A. Zunino, M. Campo). Computer Languages, Systems and Structures. Elsevier Science. ISSN: 1477-8424. Ed.: R. S. Ledley and B. Kalyanasundaram. Vol. 36, Num. 3, pp 288-315. 2010. DOI: doi:10.1016/j.cl.2009.12.003

ming. On the other hand, the same sequential application can be seamlessly and easily ported to different parallel libraries and environments.

The cornerstones of EasyFJP are an heuristic algorithm that automatically spots the points in which join primitives must be introduced, and a policy-based support for FJP –currently supporting threshold-related and cache-based optimizations– that allows users to tune the performance of the resulting parallel application without modifying its source code. We evaluated our approach by executing some CPU-intensive classic divide and conquer applications both generated via EasyFJP (with its binding to Satin) and manually implemented by using Satin. First, we compared the performance of the EasyFJP applications with respect to the Satin implementations to assess the effectiveness of our heuristic code analysis techniques. The applications were run in a local and a wide-area cluster (Section 5.1). Second, we used the former setting to parallelize and tune two benchmark applications via EasyFJP plus policies, and Satin (Section 5.2). These two evaluations involved the usage of applications with small to moderate tasks granularities. Third, we parallelized an existing implementation of the sequence alignment application by using Satin and EasyFJP and executed the resulting parallel codes in the wide-area setting. The goal of this evaluation was to test the effectiveness of EasyFJP when dealing with more complex applications and large task granularities.

With respect to the first evaluation, the EasyFJP applications performed closely to the Satin implementations in the local cluster, while outperformed Satin in the wide-area cluster. These positive results cannot be generalized, however they are very encouraging since adding parallelism to an application with EasyFJP is almost independent of the targeted parallel library. In other words, these results suggest that our generic heuristic for inserting fork and join primitives, instantiated in the evaluation through the Satin parallel library, may lead to parallel software whose performance is competitive with respect to manually using a parallel library to parallelize sequential codes. Nevertheless, we are planning to conduct further experiments with other applications and parallel libraries. Likewise, the comparisons between EasyFJP applications and policies versus applications manually optimized with Satin resulted in very acceptable execution time overheads and some performance gains for threshold and memoization policies, respectively. This is also encouraging, as it confirms that supporting policies to effectively and non-invasively tune EasyFJP parallel applications is feasible from a practical point of view. Finally, the execution of the sequence alignment application yielded as a result good performances and speedups in favor of EasyFJP, which suggest that our framework is also applicable to real-world applications.

It is worth emphasizing that EasyFJP does not aim at replacing explicit parallelism. Instead, our utmost goal is to target users who need to rapidly turn their sequential codes into parallel ones, but deal with as few parallel programmming details as possible. EasyFJP addresses this requirement by handling parallelism at a high level of abstraction, this is, automates the process of obtaining an FJP application in a backend--independent way, and provides mechanisms that capture common patterns for tuning the performance of FJP applications. However, it is a well-known fact in parallel programming that such a high-level, implicit approach may produce applications whose performance is below the levels that can be obtained by using explicit parallelism [35]. In the context of our work, this means that using EasyFJP neither necessarily leads to exploiting parallelism in an optimal way nor replaces backend knowledge. In fact, in the experiments, the applications are configured to use a particular task scheduling algorithm of Satin depending on the experimental setting (i.e. local-area or wide-area cluster). EasyFJP in turn captures recurrent parallel synchronization and tuning patterns present in FJP applications while achieving competitive performance with respect to manual parallelism, as suggested by the positive experimental evidence reported in the paper.

We are extending our work in several directions. We are working on builders for other parallel libraries (currently Doug Lea's framework and GridGain) and other distributed infrastructures (currently Terracotta and GMAC) for supporting memoization at the caching level. In this way, users will be allowed to select the target technologies that best suit their needs when generating parallel applications. Particularly, the inclusion of builders for parallel libraries such as Doug Lea's framework and GridGain, which are able to exploit multi-core individual machines and multi-core clusters, respectively, will create the opportunity for a follow-up study of the interrelation of the EasyFJP framework and such kind of execution environments. Nevertheless, in this paper we provided a consistent, rigorous evaluation of EasyFJP through the parallelization of a representative set of applications under a local-area and a wide-area cluster to evidence the applicability of the approach.

Moreover, despite the policies illustrated in the paper were implemented in pure Java, we are developing

An Approach for Non-Intrusively Adding Malleable Fork/Join Parallelism into Ordinary JavaBeans Compliant Applications. (C. Mateos, A. Zunino, M. Campo). Computer Languages, Systems and Structures. Elsevier Science. ISSN: 1477-8424. Ed.: R. S. Ledley and B. Kalyanasundaram. Vol. 36, Num. 3, pp 288-315. 2010. DOI: doi:10.1016/j.cl.2009.12.003

a policy support based on the Java scripting API [75], which allows developers to run scripts implemented in various languages (e.g. BeanShell, Python, Ruby, etc.) from within a Java application. Using these languages will produce more compact policy code, and will allow users to avoid the tedious code-compile-run task sequence when tuning applications, thus enabling for more flexible application tuning scenarios. We are also investigating and implementing more sophisticated policies in order to offer developers a wide variety of general-purpose optimization rules to regulate the amount of parallelism of their applications for typical situations. Alternatives include policies that take into account aspects such as task granularities, amount of network communication, historic executions, etc. From a conceptual perspective, these "smarter" policies for controlling the (non-functional) tuning behavior of application components, together with the flexibility inherent to scripting languages to dynamically change the rules that govern such behavior, smoothly align with the recent trend of Autonomic Computing (AC) [8]. AC represent a vast number of systems –including some of approaches mentioned before, namely GCM and K-Components– that support the construction of self-managed distributed applications. Particularly, a class of AC systems are those that enable dynamic adaptations of applications through the provision of rules external to the application code that can be added, modified or removed at runtime.

The policy concept can potentially provide a solution to some related problems that arise as a consequence of the transformational approach to parallelism of EasyFJP, which allows an individual user application to be easily ported to several backends, but this forces the underlying task execution engine to rely on randomized schedulers to handle the runtime task tree resulting from executing an FJP application. This in turn may lead to suboptimal cache usage or task-to-processor mapping in many cases. With respect to the former problem, depth-first task execution schemes allow for better cache exploitation and reduced number of runtime tasks (with the consequent savings in terms of allocated task space), however breadth-first schemes commonly maximize parallelism. To tackle down this problem, modern backends such as Intel® TBB [69] rely on a hybrid scheme of task processing. Here, EasyFJP policies could be used to indicate the underlying scheduler what scheme to use, provided the target backend offers this flexibility. With respect to the latter problem, recent HPC platforms such as Google's MapReduce [55] and GridGain [41] support the concept of *data affinity*, this is, ensuring that a group of related cache entries is contained within a single cache partition (e.g. a cluster node). Then, tasks are mapped to processors not only based on their computational requirements but also on the required data. Again, we could use policies to allow developers to control some aspects of these mapping based on the nature of the involved tasks. In summary, we will study whether the policy framework can be extended to support the above notions.

Finally, we are planning to evaluate EasyFJP from a software engineering perspective. A recent study in the context of the CO_2P_3S [59] parallel pattern-based language has shown that generative programming approaches contribute to increment the productivity of parallel software development [60]. Therefore, we could evaluate the implications of using EasyFJP and its parallelization model for implementing parallel software compared to generative programming approaches like CO_2P_3S and the GAUGE Grid system [45] by conducting a controlled case study that takes into account productivity as well as human factors.

Acknowledgments

We thank Cristian Clasadonte for his good predisposition and valuable help managing the computing infrastructure used for conducting the experiments described in this paper. We also deeply thank the anonymous reviewers for their helpful comments and suggestions to improve the quality of the paper. We acknowledge the financial support provided by ANPCyT through grants PAE-PICT 2007-02311 and PAE-PICT 2007-02312.

References

[1] J. Al-Jaroodi, N. Mohamed, H. Jiang, D. Swanson, Algorithms and Tools for Parallel Computing on Heterogeneous Clusters, chap. An Overview of Parallel and Distributed Java for Heterogeneous Systems: Approaches and Open Issues, Nova Science Publishers, Hauppauge, NY, USA, 2007, pp. 1–14.

An Approach for Non-Intrusively Adding Malleable Fork/Join Parallelism into Ordinary JavaBeans Compliant Applications. (C. Mateos, A. Zunino, M. Campo). Computer Languages, Systems and Structures. Elsevier Science. ISSN: 1477-8424. Ed.: R. S. Ledley and B. Kalyanasundaram. Vol. 36, Num. 3, pp 288-315. 2010. DOI: doi:10.1016/j.cl.2009.12.003

[2] E. Alba, C. Blum, P. Asasi, C. Leon, J. A. Gomez, Optimization Techniques for Solving Complex Problems, Parallel and Distributed Computing, Wiley Publishing, 2009.

[3] M. Aldinucci, S. Campa, M. Danelutto, M. Vanneschi, P. Kilpatrick, P. Dazzi, D. Laforenza, N. Tonellotto, Behavioural skeletons in GCM: Autonomic management of Grid components, in: 16th Euromicro Conference on Parallel, Distributed and Network-Based Processing (PDP '08), Toulouse, France, IEEE Computer Society, Washington, DC, USA, 2008, pp. 54–63.

[4] M. Aldinucci, M. Danelutto, Skeleton-based parallel programming: Functional and parallel semantics in a single shot, Computer Languages, Systems & Structures 33 (3-4) (2007) 179–192.

[5] M. Aldinucci, M. Danelutto, P. Dazzi, Muskel: An expandable skeleton environment, Scalable Computing: Practice and Experience 8 (4) (2007) 325–341.

[6] E. Allen, D. Chase, J. Hallett, V. Luchangco, J.-W. Maessen, S. Ryu, G. Steele, S. Tobin-Hochstadt, The Fortress language specification, `http://www.cis.udel.edu/~cavazos/cisc879/papers/fortress.pdf` (Mar. 2007).

[7] I. Alshabani, R. Olejnik, B. Toursel, Service oriented adaptive Java applications, in: 3rd Workshop on Middleware for Service Oriented Computing (MW4SOC '08), Leuven, Belgium, ACM Press, New York, NY, USA, 2008, pp. 43–48.

[8] A. Andrzejak, A. Reinefeld, F. Schintke, T. Schütt, Future Generation Grids, chap. On Adaptability in Grid Systems, CoreGRID series, Springer US, USA, 2006, pp. 29–46.

[9] J. Archuleta, W.-C. Feng, E. Tilevich, A pluggable framework for parallel pairwise sequence search, in: 29th Annual International Conference of the IEEE - Engineering in Medicine and Biology Society (EMBS '07), Lyon, France, IEEE Service Center, USA, 2007, pp. 127–130.

[10] L. Baduel, F. Baude, D. Caromel, A. Contes, F. Huet, M. Morel, R. Quilici, Grid Computing: Software Environments and Tools, chap. Programming, Composing, Deploying on the Grid, Springer, Berlin, Heidelberg, and New York, 2006, pp. 205–229.

[11] M. Bichler, K.-J. Lin, Service-Oriented Computing, Computer 39 (3) (2006) 99–101.

[12] A. Bik, J. Villacis, D. Gannon, Javar: A prototype Java restructuring compiler, Concurrency: Practice and Experience 9 (11) (1998) 1181–1191.

[13] W. Blochinger, C. Dangelmayr, S. Schulz, Aspect-oriented parallel discrete optimization on the Cohesion desktop Grid platform, in: 6th IEEE International Symposium on Cluster Computing and the Grid (CCGRID '06), IEEE Computer Society, Washington, DC, USA, 2006, pp. 49–56.

[14] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, Y. Zhou, Cilk: An efficient multithreaded runtime system, Parallel and Distributed Computing 37 (1) (1996) 55–69.

[15] J. M. Bull, M. E. Kambites, JOMP-an OpenMP-like interface for Java, in: ACM Conference on Java Grande (JAVA '00), San Francisco, CA, USA, ACM Press, New York, NY, USA, 2000, pp. 44–53.

[16] B. Carpenter, V. Getov, G. Judd, A. Skjellum, G. Fox, MPJ: MPI-like message passing for Java, Concurrency: Practice and Experience 12 (11) (2000) 1019–1038.

[17] M. Chalabine, C. Kessler, Crosscutting concerns in parallelization by invasive software composition and aspect weaving, in: 39th Annual Hawaii International Conference on System Sciences (HICSS '06), Kauai, Hawaii, vol. 9, IEEE Computer Society, Los Alamitos, CA, USA, 2006, p. 214b.

[18] B. Chamberlain, D. Callahan, H. Zima, Parallel programmability and the Chapel language, International Journal of High Performance Computing Applications 21 (3) (2007) 291–312.

An Approach for Non-Intrusively Adding Malleable Fork/Join Parallelism into Ordinary JavaBeans Compliant Applications. (C. Mateos, A. Zunino, M. Campo). Computer Languages, Systems and Structures. Elsevier Science. ISSN: 1477-8424. Ed.: R. S. Ledley and B. Kalyanasundaram. Vol. 36, Num. 3, pp 288-315. 2010. DOI: doi:10.1016/j.cl.2009.12.003

[19] H. N. A. Chan, A. J. Gallagher, A. S. Goundan, Y. L. W. Au Yeung, A. W. Keen, R. A. Olsson, Generic operations and capabilities in the JR concurrent programming language, Computer Languages, Systems & Structures 35 (3) (2009) 293–305.

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

[21] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, V. Sarkar, X10: An object-oriented approach to non-uniform cluster computing, ACM SIGPLAN Notices 40 (10) (2005) 519–538.

[22] S. Chiba, M. Nishizawa, An easy-to-use toolkit for efficient Java bytecode translators, in: Generative Programming and Component Engineering, Lecture Notes in Computer Science, Springer, Berlin / Heidelberg, 2003, pp. 364–376.

[23] CoreGRID NoE, Deliverable D.PM.04, basic features of the Grid Component Model (assessed), http://www.coregrid.net/mambo/images/stories/Deliverables/d.pm.04.pdf (last accessed September 2009) (2007).

[24] C. A. da Silva Cunha, J. L. F. Sobral, M. P. Monteiro, Reusable aspect-oriented implementations of concurrency patterns and mechanisms, in: R. Filman (ed.), 5th International Conference on Aspect-Oriented Software Development, Aspect-Oriented Software Development, ACM Press, New York, NY, USA, 2006, pp. 134–145.

[25] J. S. Danaher, I. A. Lee, C. E. Leiserson, Programming with exceptions in JCilk, Science of Computer Programming, Special Issue on Synchronization and Concurrency in Object-Oriented Languages 63 (2) (2006) 147–171.

[26] M. Danelutto, M. Aldinucci, Algorithmic skeletons meeting Grids, Parallel Computing 32 (7) (2006) 449–462.

[27] C. Dangelmayr, W. Blochinger, Aspect-oriented component assembly - a case study in parallel software design, Software: Practice and Experience 39 (9) (2009) 807–832.

[28] M. Denker, S. Ducass, Éric Tanter, Runtime bytecode transformation for Smalltalk, Computer Languages, Systems & Structures 32 (2-3) (2006) 125–139.

[29] J. Dongarra, D. Walker, MPI: A standard Message Passing Interface, Supercomputer 12 (1) (1996) 56–68.

[30] J. Dowling, V. Cahill, Self-managed decentralised systems using K-components and collaborative reinforcement learning, in: 1st ACM SIGSOFT Workshop on Self-Managed Systems (WOSS '04), Newport Beach, California, ACM Press, New York, NY, USA, 2004, pp. 39–43.

[31] T. El-Ghazawi, L. Smith, UPC: Unified Parallel C, in: 2006 ACM/IEEE Conference on Supercomputing (SC '06), Tampa, Florida, USA, ACM Press, New York, NY, USA, 2006, p. 27.

[32] B. Fitzpatrick, Distributed caching with memcached, Linux Journal 2004 (124) (2004) 5.

[33] I. Foster, The Grid: Computing without bounds, Scientific American 288 (4) (2003) 78–85.

[34] I. Foster, C. Kesselman, The Grid 2: Blueprint for a New Computing Infrastructure, Morgan-Kaufmann Publishers Inc., San Francisco, CA, USA, 2003.

[35] V. W. Freeh, A comparison of implicit and explicit parallel programming, Journal of Parallel and Distributed Computing 34 (1) (1996) 50–65.

[36] M. Frigo, C. E. Leiserson, K. H. Randall, The implementation of the Cilk-5 multithreaded language, SIGPLAN Notices 33 (5) (1998) 212–223.

An Approach for Non-Intrusively Adding Malleable Fork/Join Parallelism into Ordinary JavaBeans Compliant Applications. (C. Mateos, A. Zunino, M. Campo). Computer Languages, Systems and Structures. Elsevier Science. ISSN: 1477-8424. Ed.: R. S. Ledley and B. Kalyanasundaram. Vol. 36, Num. 3, pp 288-315. 2010. DOI: doi:10.1016/j.cl.2009.12.003

[37] T. Gautier, R. Revire, J. L. Roch, Athapascan: An API for asynchronous parallel programming, Tech. Rep. RT-0276, INRIA (2003).

[38] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, V. Sunderam, PVM Parallel Virtual Machine, A User's Guide and Tutorial for Networked Parallel Computing, MIT Press, Cambridge, MA, USA, 1994.

[39] O. Gotoh, An improved algorithm for matching biological sequences, Journal of Molecular Biology 162 (3) (1982) 705–708.

[40] P. Gotthelf, A. Zunino, C. Mateos, M. Campo, GMAC: An overlay multicast network for mobile agent platforms, Journal of Parallel and Distributed Computing 68 (8) (2008) 1081–1096.

[41] GridGain Systems, The GridGain Open Cloud Platform, http://www.gridgain.com (2009).

[42] N. Gustafsson, Axum: Language overview (v0.8), http://download.microsoft.com/download/B/D/5/BD51FFB2-C777-43B0-AC24-BDE3C88E231F/Axum%20Language%20Spec.pdf (last accessed May 2009) (2009).

[43] P. Hatcher, M. Reno, G. Antoniu, L. Bouge, Cluster computing with Java, Computing in Science and Engineering 7 (2) (2005) 34–39.

[44] M. Haustein, K.-P. Lohr, JAC: Declarative Java concurrency, Concurrency and Computation: Practice and Experience 18 (5) (2006) 519–546.

[45] F. Hernández, P. Bangalore, J. Gray, Z. Guan, K. Reilly, GAUGE: Grid Automation and Generative Environment, Concurrency and Computation: Practice and Experience 18 (10) (2006) 1293–1316.

[46] B. Hughes, Building computational Grids with Apple's Xgrid middleware, in: 2006 Australasian Workshops on Grid computing and e-Research (ACSW Frontiers '06), Hobart, Tasmania, Australia, Australian Computer Society, Inc., Darlinghurst, Australia, Australia, 2006, pp. 47–54.

[47] IST Programme of the European Commission, GridCOMP - Home, http://gridcomp.ercim.org (last accessed October 2009) (2009).

[48] java.net, java2xml, https://java2xml.dev.java.net (last accessed May 2009) (2004).

[49] R. Johnson, J2EE development frameworks, Computer 38 (1) (2005) 107–110.

[50] A. Jugravu, T. Fahringer, JavaSymphony, a programming model for the Grid, Future Generation Computer Systems 21 (1) (2005) 239–246.

[51] A. Kaminsky, Parallel Java: A unified API for shared memory and cluster parallel programming in 100Distributed Processing Symposium (IPDPS '07), Long Beach, CA, USA, IEEE Computer Society, Washington, DC, USA, 2007, pp. 1–8.

[52] H. Kasim, V. March, R. Zhang, S. See, Survey on parallel programming model, in: Network and Parallel Computing, vol. 5245 of Lecture Notes in Computer Science, Springer, Berlin / Heidelberg, 2008, pp. 266–275.

[53] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier, J. Irwin, Aspect-oriented programming, in: M. Akşit, S. Matsuoka (eds.), 11th European Conference on Object-Oriented Programming (ECOOP '97), vol. 1241 of Lecture Notes in Computer Science, Springer, New York, NY, USA, 1997, pp. 220–242.

[54] K. Kurowski, W. de Back, W. Dubitzky, L. Gulyás, G. Kampis, M. Mamonski, G. Szemes, M. Swain, Complex system simulations with QosCosGrid, in: Computational Science - 9th International Conference (ICCS '09), Baton Rouge, LA, USA, vol. 5544 of Lecture Notes in Computer Science, Springer, Berlin / Heidelberg, 2009, pp. 387–396.

An Approach for Non-Intrusively Adding Malleable Fork/Join Parallelism into Ordinary JavaBeans Compliant Applications. (C. Mateos, A. Zunino, M. Campo). Computer Languages, Systems and Structures. Elsevier Science. ISSN: 1477-8424. Ed.: R. S. Ledley and B. Kalyanasundaram. Vol. 36, Num. 3, pp 288-315. 2010. DOI: doi:10.1016/j.cl.2009.12.003

[55] R. Lämmel, Google's MapReduce programming model — revisited, Science of Computer Programming 68 (3) (2007) 208–237.

[56] D. Lea, The java.util.concurrent synchronizer framework, Science of Computer Programming 58 (3) (2005) 293–309.

[57] E. A. Lee, The problem with threads, Computer 39 (5) (2006) 33–42.

[58] W.-M. Lin, Performance modeling and analysis of correlated parallel computations, Parallel Computing 34 (9) (2008) 521–538.

[59] S. MacDonald, J. Anvik, S. Bromling, J. Schaeffer, D. Szafron, K. Tan, From patterns to frameworks to parallel programs, Parallel Computing 28 (12) (2002) 1663–1683.

[60] S. MacDonald, K. Tan, J. Schaeffer, D. Szafron, Deferring design pattern decisions and automating structural pattern changes using a design-pattern-based programming system, ACM Transactions on Programming Languages and Systems 31 (3) (2009) 1–49.

[61] K. E. Maghraoui, T. J. Desell, B. K. Szymanski, C. A. Varela, The Internet Operating System: Middleware for adaptive distributed computing, International Journal of High Performance Computing Applications 20 (4) (2006) 467–480.

[62] C. Mateos, A. Zunino, M. Campo, Extending Movilog for supporting Web Services, Computer Languages, Systems & Structures 33 (1) (2007) 11–31.

[63] C. Mateos, A. Zunino, M. Campo, A survey on approaches to gridification, Software: Practice and Experience 38 (5) (2008) 523–556.

[64] C. Mateos, A. Zunino, M. Campo, Grid-enabling applications with JGRIM, International Journal of Grid and High Performance Computing 1 (3) (2009) 52–72.

[65] C. E. McDowell, D. P. Helmbold, Debugging concurrent programs, ACM Computing Surveys 21 (4) (1989) 593–622.

[66] R. Montanari, E. Lupu, C. Stefanelli, Policy-based dynamic reconfiguration of mobile-code applications, Computer 37 (7) (2004) 73–80.

[67] A. Morajko, T. Margalef, E. Luque, Design and implementation of a dynamic tuning environment, Journal of Parallel and Distributed Computing 67 (4) (2007) 474–490.

[68] A. Moustafa, JAligner: Open source Java implementation of Smith-Waterman, `http://jaligner.sourceforge.net` (last accessed September 2009) (2008).

[69] C. Pheatt, Intel®threading building blocks, Journal of Computing Sciences in Colleges 23 (4) (2008) 298–298.

[70] D. Sallings, spymemcached: Java client for Memcached, `http://code.google.com/p/spymemcached` (last accessed May 2009) (2009).

[71] L. Silva, R. Buyya, High Performance Cluster Computing: Programming and Applications, chap. Parallel Programming Models and Paradigms, Prentice Hall, NJ, USA, 1999.

[72] J. Sobral, A. Proença, Enabling JaSkel skeletons for clusters and computational Grids, in: IEEE International Conference on Cluster Computing, Austin, Texas, USA, IEEE Computer Society, Los Alamitos, CA, USA, 2007, pp. 365–371.

[73] Sourceforge.net, JCGrid, `http://jcgrid.sourceforge.net` (2004).

[74] Sourceforge.net, Java Parallel Processing Framework, `http://www.jppf.org` (2009).

An Approach for Non-Intrusively Adding Malleable Fork/Join Parallelism into Ordinary JavaBeans Compliant Applications. (C. Mateos, A. Zunino, M. Campo). Computer Languages, Systems and Structures. Elsevier Science. ISSN: 1477-8424. Ed.: R. S. Ledley and B. Kalyanasundaram. Vol. 36, Num. 3, pp 288-315. 2010. DOI: doi:10.1016/j.cl.2009.12.003

[75] Sourceforge.net, Java Scripting API, https://scripting.dev.java.net (last accessed June 2009) (2009).

[76] Sun Microsystems, Java Management Extensions (JMX), http://java.sun.com/products/JavaManagement (last accessed June 2009) (2009).

[77] Sun Microsystems, JavaBeans Specification, http://java.sun.com/javase/technologies/desktop/javabeans/docs/spec.html (last accessed March 2009) (2009).

[78] C. Szyperski, Component technology - what, where, and how?, in: 25th International Conference on Software Engineering (ICSE '03), Portland, Oregon, USA, IEEE Computer Society, Washington, DC, USA, 2003, pp. 684–693.

[79] TATA Consultancy Services, The Wide Area Network Emulator, http://wanem.sourceforge.net (last accessed June 2009) (2009).

[80] Terracotta Inc., The Definitive Guide to Terracotta: Cluster the JVM for Spring, Hibernate and POJO Scalability, APRESS, New York, NY, USA, 2008.

[81] University of Virginia, jPVM, http://www.cs.virginia.edu/~ajf2j/jpvm.html (last accessed April 2009) (1999).

[82] W. van Heiningen, S. MacDonald, T. Brecht, Babylon: Middleware for distributed, parallel, and mobile Java applications, Concurrency and Computation: Practice and Experience 20 (10) (2008) 1195–1224.

[83] G. Wrzesinska, J. Maassen, K. Verstoep, H. E. Bal, Satin++: Divide-and-share on the Grid, in: 2nd IEEE International Conference on e-Science and Grid Computing (E-SCIENCE '06), Amsterdam, Netherlands, IEEE Computer Society, Washington, DC, USA, 2006, p. 61.

[84] G. Wrzesinska, R. van Nieuwport, J. Maassen, T. Kielmann, H. Bal, Fault-tolerant scheduling of fine-grained tasks in Grid environments, International Journal of High Performance Computing Applications 20 (1) (2006) 103–114.

[85] C.-T. Yang, T.-F. Han, H.-C. Kan, G-BLAST: A Grid-based solution for mpiBLAST on computational Grids, Concurrency and Computation: Practice Experience 21 (2) (2009) 225–255.

[86] K. Yelick, P. Hilfinger, S. Graham, D. Bonachea, J. Su, A. Kamil, K. Datta, P. Colella, T. Wen, Parallel languages and compilers: Perspective from the Titanium experience, International Journal of High Performance Computing Applications 21 (3) (2007) 266–290.

[87] C. Zambas, M. Luján, Introducing aspects to the implementation of a Java fork/join framework, in: Algorithms and Architectures for Parallel Processing, Lecture Notes in Computer Science, Springer, Berlin / Heidelberg, 2008, pp. 294–304.

[88] B.-Y. Zhang, Z.-Y. Mo, G.-W. Yang, W.-M. Zheng, Dynamic load-balancing and high performance communication in JCluster, 21th IEEE International Parallel and Distributed Processing Symposium (IPDPS '07), Long Beach, California, USA (2007) 227.

[89] H. Zhang, J. Lee, R. Guha, VCluster: A thread-based Java middleware for SMP and heterogeneous clusters with thread migration support, Software: Practice and Experience 38 (10) (2008) 1049–1071.

[90] H. Zhu, Z. Yin, Y. Ding, Java Annotated Concurrency based on the concurrent package, in: 7th International Conference on Parallel and Distributed Computing, Applications and Technologies (PDCAT '06), IEEE Computer Society, Washington, DC, USA, 2006, pp. 38–43.

Cristian Mateos (http://www.exa.unicen.edu.ar/~cmateos) received a Ph.D. degree in Computer Science from UNICEN, Tandil, Argentina, in 2008. He is a Full Teacher Assistant at the Computer Science Department of UNICEN and a research fellow of the CONICET. His thesis was on solutions to ease Grid application development and tuning through dependency injection and policies.

An Approach for Non-Intrusively Adding Malleable Fork/Join Parallelism into Ordinary JavaBeans Compliant Applications. (C. Mateos, A. Zunino, M. Campo). Computer Languages, Systems and Structures. Elsevier Science. ISSN: 1477-8424. Ed.: R. S. Ledley and B. Kalyanasundaram. Vol. 36, Num. 3, pp 288-315. 2010. DOI: doi:10.1016/j.cl.2009.12.003

Alejandro Zunino (http://alejandrozunino.co.cc) received a Ph.D. degree in Computer Science from UNICEN in 2003. He is a Full Assistant Professor at the Computer Science Department of UNICEN and a research fellow of the CONICET. He has published over 30 papers in journals and conferences.

Marcelo Campo (http://www.exa.unicen.edu.ar/~mcampo) received a Ph.D. degree in Computer Science from UFRGS, Porto Alegre, Brazil. He is a Full Associate Professor at the Computer Science Department and Head of the ISISTAN. He is also a research fellow of the CONICET. He has over 70 papers published in conferences and journals about software engineering topics.