PAL: HIGH LEVEL PARALLEL PROGRAMMING
WITH JAVA ANNOTATIONS*

M. Danelutto*, M. Pasin*, M. Vanneschi*
*Dept. Computer Science – Univ. of Pisa – Italy
*EIA/FR – Fribourg – Switzerland
marcod@di.unipi.it, marcelopasin@gmail.com, vannesch@di.unipi.it

P. Dazzi*, L. Presti*, D. Laforenza*
◦IMT Lucca (Institute for Advanced Studies) - Lucca - Italy
*ISTI/CNR – Pisa – Italy
patrizio.dazzi@isti.cnr.it, luigi.presti@imtlucca.it, domenico.laforenza@isti.cnr.it

CoreGRID Programming Model Institute, EIA/FR – Switzerland & Dept. Comp. Science – Pisa – Italy

Abstract We present a new programming model based on user annotations that can be used to transform plain Java programs into suitable parallel code that can be run on workstation clusters, networks and grids. The only user responsibility consists in decorating the methods that will eventually be executed in parallel with standard Java 1.5 annotations. Then these annotations are automatically processed and parallel byte code is derived. When the annotated program is started, it automatically retrieves the information about the executing platform and evaluates the information specified inside the annotations to transform the byte-code into a semantically equivalent multithreaded/multitask version. The results returned by the annotated methods, when invoked, are futures with a wait-by-necessity semantics. A PAL prototype has been implemented in Java, using JJPF [11] as Parallel Framework. The experiments made with the prototype are encouraging: the design of parallel applications has been greatly simplified and the performances obtained are the same of an application directly written in JJPF.

Keywords: Asynchronous method invocation, wait-by-necessity, annotations, grids.

*This work has been partially supported by Italian national FIRB project no. RBNE01KNFP GRID.it and by the FP6 Network of Excellence CoreGRID funded by the European Commission (Contract IST-2002-004265).
1. Introduction

Grid computing [18] enables the use of a (very) large number of networked processing resources equipped with suitable middleware to provide powerful platforms that can be used to support high performance computing, pervasive (global, ubiquitous) computing as well as to provide advanced “knowledge utility” environments [17]. Developing parallel/distributed applications targeting the grid is in general more complex than developing similar applications for traditional parallel architectures and workstation clusters. Besides being in charge of the whole parallel application structure as well as of all the relative communication, synchronization, mapping and scheduling structure, the programmer must also take into account that grid processing resources are often heterogeneous and that the availability of both the computing and the interconnection resources may vary in time. As the programmers usually write applications directly interacting with the middleware, the whole process is cumbersome and error prone. In the last years, several efforts have been spent to face this problem, and several approaches have been conceived to design high-level programming languages/environments that can automate most of the tasks required to implement working and efficient grid applications. Some approaches aim at providing programmers with different programming environments implementing as much as possible the “invisible grid” concept advocated by the EC Next Generation Grid Expert Group [22, 17]. As an example the Grid Component Model currently being developed within the CoreGRID Institute on Programming model [10, 24] will eventually provide the grid programmers a component based programming model and all the details and issues related to the usage of the grid as the target architecture will be dealt with in the compiler and run time tools. Other approaches offer a lower abstraction level but allow more programming freedom and guarantee a higher level of personalization. In other words, programmers can customize their applications and deal with some aspects related to the parallelism as, for example, parallelism degree and the parallel program structure (farm, pipeline, ...). The approaches belonging to this category force the programmer to structure the parallel application he wants to implement adequately. Typically, such approaches permit to separate the application “business logic” from the activities required to coordinate and to synchronize parallel processes [15, 3]. On the other side, several environments have been proposed to use more classical, low level programming paradigms on the grid. Several implementation of MPI [2] have been ported on top of different grid middlewares [20] as well as several implementations of different kinds of RPC have been designed [25, 19, 26]. However, all these approaches, while leaving the programmer a higher freedom of structuring the parallel applications in an arbitrary way, require the programmers explicitly deal with all the awkward details mentioned above.
In this work, we introduce Parallel Abstraction Layer (PAL) as a bridge between a currently popular programming model and the current parallel computers, as clusters and the grid. To avoid the problems typically present in a fully automated parallel approach [13, 4], PAL leaves to programmer the responsibility to choose which parts of code have to be computed in parallel through the insertion of non-functional requirements in the source program code. Using the information provided by programmers PAL transforms the program code into a parallel one which structure depends on the specified non-functional requirements.

A prototype of PAL has been implemented using Java. It allows to autonomically transforming the byte-code of an annotated method in a multithreaded byte-code version, suitable for multiprocessor computers and in a parallel byte-code version using the JJPF [11] parallel framework, suitable for COWs and Grids. The initial tests have shown for both versions encouraging results.

The rest of this paper is structured as follows: in Section 2, we introduce the PAL approach and in Section 3, we describe the current PAL prototype. Section 4 shows a set of preliminary experimental results. Eventually, Section 5 discusses related work and Section 6, draws conclusions and outlines future research directions.

2. Parallel Abstraction Layer (PAL)

We fully subscribe the opinion stating that “...people know the application domain and can better decompose the problem, compilers can better manage data dependence and synchronization” [21]. The approach we propose to follow in the development of parallel grid applications relies on programmer knowledge to “structure” the parallel schema of application leaving then to the compiler/run time tools the efficient implementation of the parallel schema conceived by the programmers. The general idea is outlined in Figure 1.

This is much in the sense of what’s being advocated in the algorithmical skeletons approach [9]. Actually, here we propose a general-purpose mechanism that does not require a complex application structuring by the programmer. In fact the programmer is only required to insert, in the source code, some hints that will be eventually exploited in the runtime support to implement efficient parallel/distributed execution of the application code.

As an example, these hints may consist of non-functional requirements, aka performance contract (SLA, Efficiency, Price, Reliability, Resource constraints, Software, tools, standards, parallelism degree etc.) that can be specified through annotations mechanism provided by Java and .NET [1].

Once the programmer has inserted the annotations in the source code, the run time exploits the information conveyed in the annotations to implement a
Despite the fact programmers are required to give some kind of "parallel structure" to the code directly at the source code level, as it happens in the algorithmical skeleton case, the approach discussed in this work presents at least three additional advantages. First, annotations can be ignored and the semantics of the original sequential code is preserved. This means that the programmer application code can be run through a classical compiler/interpreter suite and debugged using normal debugging tools. Second, annotations are processed at load time, typically exploiting reflection properties of the hosting language. As a consequence, while handling annotations, a bunch of knowledge can be exploited which is not available at compile time (kind of machines at hand, kind of interconnection network, etc.) and this can lead to more efficient parallel implementations of the user application. Third, the knowledge concerning the kind of target architecture can be exploited leading to radically diverse implementation of the very same user code. As an example, if the run time can figure out that the target architecture where the program is running happens to be a grid, it can transform the code in such a way very coarse grain parallelism is exploited. On the other hand, in case the run time figures out that user asked to execute the code on a SMP target, a more efficient, possibly finer grain, multithreaded version of the code can be produced as the result of the annotation handling.
In order to experiment the feasibility of the proposed approach, we considered the languages that natively support code annotations. Both Java and .NET frameworks provide an annotation mechanism. They also provide an intermediate language (IL) \[31\], portable among different computer architecture (compile once – run everywhere), and holding some information typically only available at source code level (e.g. code annotations) that can be used in the runtime for optimization purposes.

The optimization we propose consists in the automatic restructuring of the application in order to exploit the application parallelism with respect to programmer’s annotations (non-functional application requirements). The transformation process is done at load time, that is at the time we have all the information we need to optimize the restructuring process with respect to the available parallel tools and underlying resources. The code transformation works at IL level thus it does not need that the application source code is sent on target architecture. Furthermore, IL transformation introduces less overhead than the source code one plus the subsequent compilation.

More in detail, we designed a Parallel Abstraction Layer (PAL) filling the gap between the traditional and the parallel programming metaphor. PAL is a generative \[23\] metaprogramming engine, which gathers, at load time, all information on available parallel tools and computational resources. Then, it analyzes the IL code looking for programmer annotations (non-functional requirements) in order to directly transform sequential IL code to parallel code, while satisfying the performance contracts supplied by the programmers through the annotations in the source code. The structure of the new IL code depends on the selected parallel framework and by the value of some non-functional requirements.

PAL exploits the parallelism by asynchronously executing parts of the original code. The parts to be executed asynchronously are individuated by the user annotations. In particular, we used Java and therefore the more natural choice was to individuate method calls as the parts to be asynchronously executed. PAL translates the IL codes of the “parallel” part by structuring them as needed by the parallel tools/libraries available on the target architecture. Asynchronous execution of method code is based on the concept of future \[7–8\]. When a method is called asynchronously it immediately returns a future, that is a stub “empty” object. The caller can then go on with its own computations and use the future object just when the method call return value is actually needed. If in the meanwhile the return value has already been computed, the call to reify the future succeeds immediately, otherwise it blocks until the actual return value is computed and then returns it.

PAL programmers must simply put a @Parallel annotation (possibly enriched with some other non-functional requirements, such as the required parallelism degree, as an example) on the line right before method declaration to mark that method as a candidate for asynchronous execution. This allows
keeping applications much like normal sequential applications, actually. Programmers may simply run the application through standard Java tools to verify it is functionally correct. The PAL approach also avoids the proliferation of source files and classes, as it works transforming IL code, but raises several problems related to data sharing management. As an example, methods annotated with a @Parallel cannot access class fields: they may only access their own parameters and the local method variables. This is due to the impossibility to intercept all the accesses to the class fields, actually. Then PAL autonomically performs at load time activities in order to render the execution of the PAL-annotated methods asynchronous and parallel and to manage any consistency related problems, without any further programmer intervention.

3. A PAL prototype

We have implemented a PAL prototype in Java 1.5, as Java provides a manageable intermediate language (Java byte-code [30]) and natively supports code annotations, since version 1.5. Furthermore, it owns all the properties needed by our approach (type safety, security, etc.). The prototype works taking the program byte-code as input and transforming it in a parallel or multithreaded byte-code (see Fig. 1). In order to do this it uses ASM [5]: a Java byte-code manipulation framework.

The current prototype accepts only one kind of attribute to the @Parallel annotations, a parDegree denoting the number of processing elements to be used for the method execution. PAL uses such information to make a choice between the multithreaded and distributed version. This choice is driven by the number of processors/cores available on the host machine: if the machine owns a sufficient number of processors the annotated user byte-code is transformed in a semantically equivalent multithreaded version. Otherwise PAL chooses to transform user byte-code in a semantically equivalent parallel version that uses several networked machines to execute the program.

Concerning this second case, PAL only produces parallel code compliant with the JJPF framework [11–12], at the moment. JJPF is a framework, based on Jini Technology, designed to provide programmers with an environment supporting the execution of skeleton based parallel applications, providing fault-tolerance and load-balancing. PAL basically transforms code in such a way the user code relative to methods to be computed asynchronously is embedded into the code of the remote JJPF servers displaced onto the processing elements used for parallel computation of the application. Conversely, the main code invoking the @Parallel methods is used to implement the “client” code, i.e. the application the user runs on its own local machine. This application eventually will interact with the remote JJPF servers according to proper JJPF mechanisms.
and protocols. This implementation schema looks like very close to a classical master/slave implementation.

We could have used any other parallel programming framework as the PAL target. As an example, we could have used Globus toolkit. However, JJPf was more compact and required a slightly more compact amount of code to be targeted, with respect to the Globus or other grid middleware frameworks. As the principles driving the generation of the parallel code are the same both using JJPf and other grid middleware frameworks, we preferred JJPf to be able to implement a proof-of-concept prototype in a short time.

Current PAL prototype therefore accepts plain Java programs with methods annotated as \texttt{@Parallel} and generates either multithreaded parallel code or parallel code suitable for the execution on a network of workstations running Java/JINI and JJPf. It has some limitations, however. In particular, the only parameter passing semantics available for annotated methods is the \textit{deep-copy} one, and the current prototype does not allows to access the class fields from inside the annotated methods.

In order to enable the PAL features, the programmer has only to add a few lines of code. Figure 1 shows an example of PAL prototype usage, namely a program computing the Mandelbrot set. The \texttt{Mandelbrot} class uses a \texttt{@Parallel} annotation to state that all the calls to the \texttt{createLines} should be computed in parallel, with a parallelism degree equal to 16. Observe that, due to some Java limitations (see below), the programmer must specify \texttt{PFFuture} as return type, and consequently return an object of this type. \texttt{PFFuture} is a template of the PAL framework, which represents a container needed to enable the future mechanism. The type specified as argument is the original method return type. Initially, we tried to have to a more transparent mechanism for the future implementation, without any explicit Future declaration. It consisted in the load-time substitution of the return type with a PAL-type inheriting from the original one. In our idea, the PAL-type would have filtered any original type dereferentiation following the \textit{wait-by-necessity} \cite{6} semantics. Unfortunately, we had to face two Java limitations that limit the current prototype in the current solution. These limitations regard the impossibility to extend some widely used Java BCL classes (\texttt{String, Integer,...}) because they are declared \texttt{final}, and the impossibility to intercept all class field accesses.

In the \texttt{Main} class, the user just asks to transform the \texttt{Main} and the \texttt{Mandelbrot} classes with PAL, that is, to process the relevant PAL annotations and to produce an executable IL which exploits parallelism according to the features (hw and sw) of the target architecture where the \texttt{Main} itself is being run.
4. Experimental results

To validate our approach we ran some experiments with the current prototype. The tests were made for both versions: multithreaded and parallel. For the former we have used, as test bed, a bi-processors workstation (Intel Xeon 2Ghz, Linux kernel 2.6), for the latter, instead, a blade cluster (24 machines single PentiumIII-800Mhz processor with multiple Fast Ethernet network, Linux kernel 2.4). For both versions, our test application was a fractal image generator, which computes sections of the Mandelbrot set.

To study in more detail the behavior of the transformed version in several contexts, we ran the fractal generator setting different combinations of resolution (600x400, 1200x800, 2400x1600) and task computational weights (starting from 1 up to 40 lines at time). The transformed multithreaded version has been executed only with parDegree value equals to 1 or 2 (we used a bi-processor test bed). Nevertheless, the multithreaded experiments achieved promising results, as the registered efficiency with parallel degree 2 is about 1, for all the combination (resolution and compute lines). Since in a multicore solution we have a lower communication impact than in a COW or grid solution, we can point out that this performance should be easily maintained with symmetric multiprocessors with even larger processing element set.

When the very same source code is used on a distributed workstation network with JJPF we achieved performances definitely close to the ones we achieved with hand written JJPF code (see Fig. 2), instead. The Figure shows the result of the experiments with an image resolution of 2400x1600, the other results obtained using different image resolutions being comparable, when a different number of processing elements were used (i.e. when different values were passed to the @Parallel(parDegree=...) annotation).
These results demonstrate that PAL performance strictly depends on the parallel tool targeted by the PAL IL transformation techniques. Actually, the overhead introduced by PAL is negligible. Nevertheless, an overhead exists because PAL offers to programmers a general metaphor that is not specialized with respect to the parallel tool used at runtime.

5. Related work

PAL offers a simple yet expressive technique for parallel programming. Exploiting “runtime compilation” it adapts the executable code to different architectures, such as shared memory multiprocessors and networked multicomputers. It does not introduce a new or different paradigm, yet allowing for parallelism at the method call level. We found in the literature a certain number of systems with similar ideas, presented as follows.

Although several different experiments exist in the so-called concurrent object-oriented languages scenario (COOLs) [28], we decided to discuss only those very similar to PAL. We concentrate on those proposed to transform a sequential object-oriented program into a concurrent one by replacing method invocations with asynchronous calls. This is based on the perception that parallelism can be easily extracted from sequential code without modification, and without changing the sequential semantics. The wait for return values can be postponed to the value next usage, eventually using future objects. All systems presented below share (at least some of) these characteristics with PAL.

Java made popular the remote method invocation (RMI) for interaction between objects in disjoint memories. The same properties that apply for parallelizing sequential local calls apply for remote ones, with the advantage that remote calls do not rely on shared memory. Parallelizing RMIs scales much better than local calls, as the number of parallel tasks is not limited by the number of local processors. This led to many implementations of asynchronous RMIs.

ProActive is a popular advocate of asynchronous RMIs [16]. It offers a primitive class that should be extended to create remote callable active objects, as well as a runtime system to remotely instantiate this type of objects. Any call to an active object is done asynchronously, and values are returned using future objects. Compilation is completely standard, but instantiation must be done supplying the new object location. All active objects must descend from the primitive active object class, so existing code must be completely encapsulated to become active, as there is no multiple inheritance in Java. Although concurrency is available through asynchronous calls, scalable parallelism is obtained creating several distributed objects, instead of calling several concurrent methods, which is not always a natural way of structuring the parallelism.
Some other systems, at different levels, offer asynchronous remote method calls, like JavaParty [29] and Ibis [32]. They are placed at a lower level with respect to PAL and are more concerned in the RMI performance and asynchrony itself. Usually they offer a good replacement for the original RMI system, either simplifying object declaration or speeding up the communication. Both rely on specific compilers to generate code, although Ibis generate standard JVM byte-code that could be executed anywhere.

6. Conclusion and future works

We proposed a new technique for high level parallel programming based on the introduction of a Parallel Abstraction Layer (PAL). PAL doesn’t introduce a new parallel programming model, but actually exploits the programmer knowledge provided through annotations to restructure the application once the available target parallel framework is known. The restructuring process is driven by the analysis of the non-functional requirements introduced with code annotations. This process is executed at load time directly at intermediate language level. This allows to obtain at the right time all the information needed to parallelize the applications with respect to the parallel tools available on the executing environment and to the user supplied non-functional requirements. Load time transformation allows to hide most of parallelization issues as well and to parallelize method calls.

We developed a PAL Java prototype and we used it to perform some experiments. The results are very encouraging and show that the overhead introduced by PAL is negligible, while keeping the programmer effort to parallelize the code negligible. Nevertheless, the current prototype has some limitations. The non-functional requirements are limited to the possibility to indicate the parallelism degree, the parameter passing semantic to PAL-annotated method is limited to deep-copy and the class fields are not accessible from PAL-annotated methods. Finally, the programmer has to include an explicit dereferentiation of objects returned by PAL-annotated methods.

Completing PAL design includes further development on the prototype, by enriching it with some other features. In particular, we think to support distributed field access from inside PAL-annotated methods and we will try to supply a greater variety of parameter passing semantics in PAL-annotated method, which is fundamental to provide a larger programming freedom. In the near future we also want to increment the set of available non-functional requirements that can be specified inside @Parallel annotation, and to add PAL the ability to generate code for different parallel frameworks, including plain Globus grids. Last but not least, we’re interested to merge the PAL experience with similar research performed at our Dept. by other people in the .NET (Mono [27]) framework [14].
References


