PAL: EXPLOITING JAVA ANNOTATIONS FOR PARALLELISM*

M. Danelutto[⋆], M. Pasin[⋆,•], M. Vanneschi[⋆]
[⋆] *Dept. Computer Science – Univ. of Pisa – Italy & CoreGRID Programming Model Institute*
[•] *EIA/FR – Fribourg – Switzerland*
marcod@di.unipi.it, marcelopasin@gmail.com, vannesch@di.unipi.it

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

Abstract We discuss how Java annotations can be used to provide the meta information needed to automatically transform plain Java programs into suitable parallel code that can be run on workstation clusters, networks and grids. Programmers are only required to decorate 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 or multitask version, depending on the target architecture features. The results returned by the annotated methods, when invoked, are futures with a wait-by-necessity semantics.
A PAL (*Parallel Abstraction Layer*) prototype exploiting the annotation based parallelizing approach has been implemented in Java. PAL targets JJPF, an existing, skeleton based, JAVA/JINI programming environment, 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, skeletons, 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 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 (*GCM*) currently being developed within the CoreGRID Institute on Programming model [10, 25] will eventually provide the grid programmers a component based programming model where 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 allow the application "business logic" to be separated 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 middleware [20] as well as several implementations of different kinds of RPC have been designed [26, 19, 27]. 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.

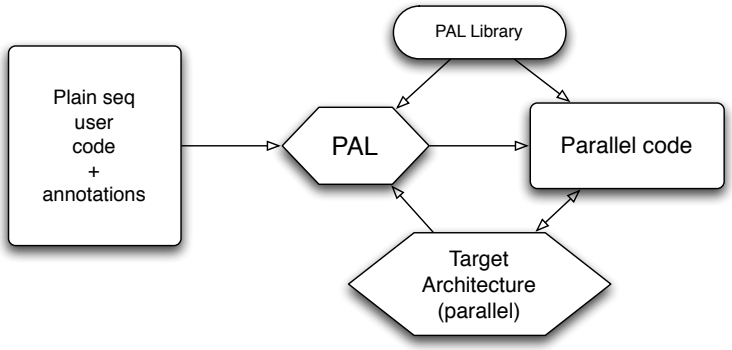

Figure 1. PAL approach overview

In this work, we introduce Parallel Abstraction Layer (PAL) as a bridge between a currently popular programming model and the typical current parallel computer architectures, such 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 (a Java/Jini Parallel Framework, [11]) parallel programming environment, targeting both clusters/networks of workstations and grids. The initial tests have shown for both versions encouraging results.

2. Parallel Abstraction Layer (PAL)

We fully subscribe the opinion "*...people know the application domain and can better decompose the problem, compilers can better manage data dependence and synchronization*" [21]. Our approach to parallel grid programming relies on programmer knowledge to "structure" the parallel schema of application and then to the compiler/run time tool ability to efficiently implement the parallel schema conceived by the programmer. The general idea is outlined in Figure 1.

This is much in the sense of what's being advocated in the algorithmic skeletons approach [9]. Actually, here we propose a general-purpose mechanism that does not require complex application structuring by the programmer. In

```
public class Mandelbrot{
   public void paint(GraphicsContext gcont) {
      // computing image size
      ...
       Vector<PFFuture<Vector<Vector<Integer>>>> man =
            new Vector<PFFuture<Vector<Vector<Integer>>>>(numOfLines);

      for(int i=0;i<numOfLines;i++)
            man.add(createLines(...);
      ...
 }

 @Parallel(parDegree=16)
 public PFFuture<Vector<Vector<Integer>>> createLines (params ...){

        Vector<Vector<Integer>> v = new Vector<Vector<Integer>>();

        // compute points ...
        for (int i = 0; i<cls; i++) {
              ...
        v.add(point);
        }
        return new PFFuture<Vector<Vector<Integer>>>(v);
 }
}

public class Main {
      ...
      public static void main(String[] args) {
            Class [] toBeTransformed = new Class[2];
            toBeTransformed[0] = Main.class;
            toBeTransformed[1] = Mandelbrot.class;
            PAL.transform(toBeTransformed,args);
            Mandelbrot mBrot = new Mandelbrot();
             BufferedImage bi = new BufferedImage(2400,1600,TYPE_INT_BGR);
            mBrot.paint(GraphicsEnvironment.getLocalGraphicsEnvironment().createGraphics(bi));

      }
}
```

Figure 2. Sample code using PAL

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.

These hints may consist of non-functional requirements. As an example, performance contracts (SLA, Efficiency, Price, Reliability, Resource constraints, Software, tools, standards, parallelism degree etc.) can be specified through the annotation mechanisms provided by both 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 parallel version of the program running on top of the target parallel/distributed architecture.

The programmers are required to give some kind of "parallel structure" to the code directly at the source code level, as it happens in the algorithmic skeleton case. However, 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) [32], 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 in general fewer overheads than the source code transformations followed by re-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 [24] 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) directly transforms the sequential IL code to the parallel code, satisfying in the meanwhile 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 on the presence and/or 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 similar to 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 aimed at achieving the asynchronous and parallel execution of the PAL-annotated methods and at managing 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 [31]) 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. 2). 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` annotation: 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 byte-code directly compiled from user code is transformed in a semantically equivalent multithreaded version. Otherwise PAL chooses to transform the compiled 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 some code suitable to be run on the remote JJPF servers displaced onto the processing elements. 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. Method call parameters, the input data for the code to be executed asynchronously, are packaged in a "task". When a server receives a task to be computed, it removes its server-descriptor from the processing elements available for JJPF. When the task computation is completed the server re-inserts its descriptor from the available ones. In other words, when a annotated method is called an empty future is immediately returned, a "task" is generated and it is inserted into the JJPF queue; eventually it is sent to one of the available processing element, which remove itself from the available resources, computes the task and returns the result that JJPF finally put inside the proper future. 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 `@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 *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 2 shows an example of PAL prototype usage, namely a program computing the Mandelbrot set. The `Mandelbrot` class uses a `@Parallel` annotation to state that all the `createLines` calls 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 `PFFuture` as return type, and consequently return an object of this type. `PFFuture` is a template defined by the PAL framework. It 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 *wait-by-necessity* [6] semantics. Unfortunately, we had to face two Java limitations that limit the current prototype to the current solution. These limitations regard the impossibility to extend some widely used Java BCL classes (String, Integer,...) because they are declared `final`, and the impossibility to intercept all class field accesses.

In the `Main` class, the user just asks to transform the `Main` and the `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 `Main` itself is being run.

4. Experimental results

To validate our approach we ran some experiments with the current prototype. We run tests were covering both cases: multithreaded and parallel transformations. In the former case, we used, as test bed, a hyper-threading bi-processors workstation (Intel Xeon 2Ghz, Linux kernel 2.6). In the latter case, instead, we used a blade cluster (24 machines single PentiumIII-800Mhz

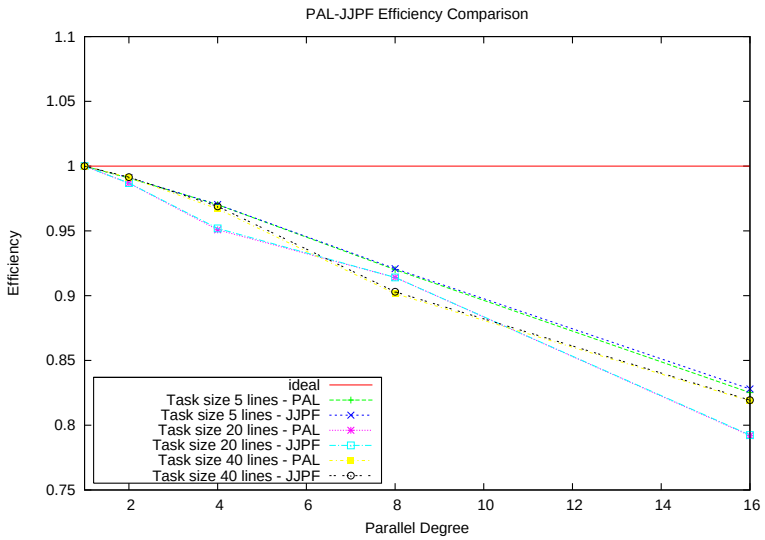

Figure 3. Mandelbrot computation: efficiency comparison with different image resolution, processing element number and task computational weight.

processor with multiple Fast Ethernet network, Linux kernel 2.4). For both cases, our test application was a fractal image generator, which computes sections of the Mandelbrot set. We picked up Mandelbrot as it is a very popular benchmark for embarrassingly parallel computation. PAL addresses exactly these kinds of computations, as it only allows executing remotely methods not accessing shared (static) variables nor having any kind of side effects. On the one hand, this obviously represent a limitation, as PAL cannot compete, as an example, with other approaches supporting plain loop parallelization. On the other hand, a huge amount of embarrassingly parallel applications are executed on clusters, workstation networks and grids. Most of times, the implementation of these applications requires a significant programming effort, despite being "easy" embarrassingly parallel, far more consistent than the effort required to execute the same kind of application exploiting PAL.

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. Clearly when the task size (number of lines to compute) increases, the total number of tasks decreases. 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 (multicore) processing elements.

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. 3), instead. The Figure shows the result of the experiments with an image resolution of 2400x1600 (other results obtained using different image resolutions are comparable) when a different number of processing elements are 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, while exploiting parallelism at the method call level. We found in the literature a certain number of systems with similar ideas. However, although different experiments exist in the so-called concurrent object-oriented languages scenario (COOLs) [29], we decided to discuss only those actually very similar to PAL.

In [23] the authors propose a Java version of OpenMP giving to the programmers the possibility to specify some PRAGMAs inside comments to source code. These pragmas are eventually used by a specific java HPC compiler to transform the original program in a different one exploiting parallelism, for instance through loop-parallelization. There are three important differences between this approach and the ours one: first of all PAL works at method level making method invocations asynchronous, while the work presented by Klemm et al. mainly works at the loop-parallelization level. Another very important difference is related to the moment in which the transformation is made: this approach works at compile time starting from source-code, while PAL directly transforms the byte-code at load and run time. As a consequence, PAL may optimize its transformation choices exploiting the knowledge available on the features of the computing resources of the target execution platform. Eventually, PAL uses *java Annotations* to enrich the source code, instead the Java

version of OpenMP uses the source code comments. The former approach exploits Java basic features, in particular annotations, which type and syntax are checked by compiler, with the limitation that annotations cannot be placed everywhere in the source code. the latter solution instead is more "artificial" but it is not limited to classes, methods and class fields (as the *java Annotations*) and it can be also applied to pure Java code blocks.

If we limit the discussion to the approaches that transform a sequential object-oriented program into a concurrent one by replacing method invocations with asynchronous calls, (where parallelism can be easily extracted from sequential code without modification, without changing the sequential semantics and the wait for return values can be postponed to the next usage, eventually using future objects) the number of approaches similar to PAL is small. However, some other approaches share single points/features with our PAL approach.

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 local processors does not limit the number of parallel tasks. This led to many implementations of asynchronous RMIs. ProActive is a popular object oriented distributed programming environment supporting 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 [30] and Ibis [33]. They provide a lower level of abstraction with respect to PAL, being more concerned with the performance of RMI and efficient implementation of asynchronous mechanisms. 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 therefore be executed on any standard JVM.

6. Conclusion and future work

We propose 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 obtaining and to exploit at the right time all the information needed to parallelize the applications with respect to the parallel tools available on the target execution environment and to the user supplied non-functional requirements. A load time transformation allows hiding most of parallelization issues.

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. Eventually, the programmer has to include an explicit dereferentiation of objects returned by PAL-annotated methods.

We are currently investigating other possibilities, in order to complete the PAL design. In particular, we are considering to support distributed field access from inside PAL-annotated methods as well as to provide a larger choice 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 [28]) framework [14].

References

[1] Java specification requests 175: A metadata facility for the java programming language. http://www.jcp.org, September 2004.

[2] Mpi: A message-passing interface strandard. http://www.mpi-forum.org, 1994.

[3] M. Aldinucci, M. Danelutto, and M. Vanneschi. Autonomic QoS in ASSIST Grid-aware components. In *Euromicro PDP 2006: Parallel Distributed and network-based Processing*. IEEE, February 2006. Montbéliard, France.

[4] G. S. Almasi and A. Gottlieb. *Highly parallel computing*. Benjamin-Cummings Publishing Co., Inc., Redwood City, CA, USA, 1989.

[5] C. T. Bruneton E, Lenglet R. Asm: a code manipulation tool to implement adaptable systems, grenoble, france. Adaptable and Extensible Component Systems, Nov. 2002.

[6] D. Caromel. Service, asynchrony, and wait-by-necessity. *Journal of Object-Oriented Programming*, Nov/Dec 1989.

[7] D. Caromel and L. Henrio. *A Theory of Distributed Object*. Springer-Verlag, 2005.

[8] D. Caromel, L. Henrio, and B. Serpette. Asynchronous and deterministic objects, 2004.

[9] M. Cole. Bringing Skeletons out of the Closet: A Pragmatic Manifesto for Skeletal Parallel Programming. *Parallel Computing*, Volume 30, Number 3, pages 389–406, 2004.

[10] Programming model Institute home page, 2006. http://www.coregrid.net/mambo/content/category/3/13/261/.

[11] M. Danelutto and P. Dazzi. A java/jini framework supporting stream parallel computations. In *Proc. of Intl. PARCO 2005: Parallel Computing*, September 2005.

[12] P. Dazzi. Jjpf: a parallel programming framework based on jini. Master's thesis, University of Pisa, July 2004. *JJPF: uno strumento per calcolo parallelo con JINI*.

[13] N. DiPasquale, T. Way, and V. Gehlot. Comparative survey of approaches to automatic parallelization. In *MASPLAS'05*, April 2005.

[14] C. Dittamo. Annotation based techniques for the parallelization of sequential programs (in Italian), July 2006. Graduation thesis, Dept. Computer Science, Univ. of Pisa.

[15] J. Dünnweber and S. Gorlatch. Component-based Grid Programming using the HOC-Service Architecture. In I. H. Fujita, editor, *New Trends in Software Methodologies, Tools and Techniques*, Frontiers in Artificial Intelligence and Applications. IOS Press, 2005. ISBN 1-58603-556-8.

[16] D. C. et al. Proactive. http://proactive.objectweb.org, 1999.

[17] K. J. et al. Future for European Grids: GRIDs and Service Oriented Knowledge Utilities, January 2006. Third report of the Next Generation Grids expert group, available at `http://cordis.europa.eu/ ist/ grids/ pub-report.htm`.

[18] I. Foster and C. Kesselman. *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kauffman, 1999.

[19] GGF RPC WG home page, 2006. https://forge.gridforum.org/projects/gridrpc-wg/.

[20] MPICH-G2 home page, 2006. http://www3.niu.edu/mpi/.

[21] A. S. Grimshaw. The mentat computation model data-driven support for object-oriented parallel processing. Technical report, Dept. Comp. Science, Univ. Virginia, 28 1993.

[22] E. N. G. G. E. Group. Next Generation Grids 2 Requirements and Options for European Grids Research 2005-2010 and Beyond, July 2004. ftp://ftp.cordis.europa.eu/ pub/ ist/ docs/ ngg2_eg_final.pdf.

[23] M. Klemm, R. Veldema, M. Bezold, and M. Philippsen. A proposal for openmp for java. In *Proceedings of the International Workshop on OpenMP*, June 2006.

[24] Krzysztof Czarnecki and Ulrich W. Eisenecker. *Generative Programming - Methods, Tools, and Applications*. Addison–Wesley, June 2000.

[25] L. Henrio et al. . Proposals for a Grid Component Model. Technical Report D.PM.02, CoreGRID, December 2005.

[26] Y. Nakajima, M. Sato, T. Boku, D. Takahashi, and H. Goto. Performance Evaluation of OmniRPC in a Grid Environment. In *Proc. of SAINT2004, Workshop on High Performance Grid Computing and Networking*, pages 658–664, January 2004.

[27] Ninf: A Global Computing Infrastructure, 2006. http://ninf.apgrid.org/.

[28] Novell. Mono project. http://www.mono-project.com/, 2005.

[29] M. Philippsen. A survey of concurrent object-oriented languages. *Concurrency: Practice and Experience*, Volume 12, Number 10, pages 917–980, 2000.

[30] M. Philippsen and M. Zenger. JavaParty – transparent remote objects in Java. *Concurrency: Practice and Experience*, Volume 9, Number 11, pages 1225–1242, Nov. 1997.

[31] F. Y. Tim Lindholm. *The Java Virtual Machine Specification*. Sun Microsystems Press, second edition edition, 2004.

[32] S. Tse. Typed intermediate languages. Technical report, Dept. Comp. Science, University of Pennsylvania, 2004.

[33] R. V. van Nieuwpoort, J. Maassen, G. Wrzesinska, R. Hofman, C. Jacobs, T. Kielmann, and H. E. Bal. Ibis: a flexible and efficient java-based grid programming environment. *Concurrency and Computation: Practice & Experience*, Volume 17, Number 7-8, pages 1079–1107, 2005.