Lithium: A Structured Parallel Programming Environment in Java

M. Danelutto & P. Teti

Dept. Computer Science – University of Pisa – Italy
{Marco.Danelutto@di.unipi.it, tetipaol@libero.it}

Abstract. We describe a new, Java based, structured parallel programming environment. The environment provides the programmer with the ability to structure his parallel applications by using skeletons, and to execute the parallel skeleton code on a workstation network/cluster in a seamless way. The implementation is based on macro data flow and exploits original optimization rules to achieve high performance. The whole environment is available as an Open Source Java library and runs on top of plain JDK.

1 Introduction

The Java programming environment includes features that can be naturally used to address network and distributed computing (JVM and bytecode, multithreading, remote method invocation, socket and security handling, and, more recently, JINI, JavaSpaces, Servlets, etc. [20]). Many efforts have been performed to make Java suitable for parallel computing too. Several projects have been started that aim at providing features that can be used to develop efficient parallel Java applications on a range of different parallel architectures. Such features are either provided as extensions to the base language or as class libraries. In the former case, ad hoc compilers and/or runtime environments have been developed and implemented. In the latter, libraries are supplied that the programmer simply uses within his parallel code. As an example extensions of the JVM have been designed that allow plain Java threads to be run in a seamless way on the different processors of a single SMP machine [2]. On the other side, libraries have been developed that allow classical parallel programming libraries/APIs (such as MPI or PVM) to be used within Java programs [15, 12].

In this work we discuss a new Java parallel programming environment which is different from the environments briefly discussed above, namely a library we developed to support *structured* parallel programming, based on the *algorithmical skeleton* concept. Skeletons have been originally conceived by Cole [5] and then used by different research groups to design high performance structured parallel programming environments [3, 4, 19]. A skeleton is basically an abstraction modeling a common, reusable parallelism exploitation pattern. Skeletons can be provided to the programmer either as language constructs [3, 4] or as libraries [9, 10]. They can be nested to structure complex parallel applications.

P.M.A. Sloot et al. (Eds.): ICCS 2002, LNCS 2330, pp. 844–853, 2002.
© Springer-Verlag Berlin Heidelberg 2002

```
import lithium.*;
...
public class SkeletonApplication {
  public static void main(String [] args) {
    ...
    Worker w = new Worker();       // encapsulate seq code in a skel
    Farm f = new Farm(w);          // use it as the task farm worker
    Ske evaluator = new Ske();     // declare an exec manager
    evaluator.setProgram(f);       // set the program to be executed
    String [] hosts = {"alpha1","alpha2",
                       "131.119.5.91"};
    evaluator.addHosts(hosts);     // define the machines to be used
    for(int i=0;i<ntasks;i++)
      evaluator.setupTaskPool(task[i]);  // prepare input stream
          // this can be done in parallel with parDo() actually
    evaluator.stopStream();              // declare its end
    evaluator.parDo();             // require parallel computation
    while(!evaluator.isResEmpty()) {     // retrieve comput. results
        Object res = evaluator.readTaskPool();
        ...
    }                              // print some statistics
    System.out.println("elapsed time = "+evaluator.getElapsedTime()+
                       "\nstartup = "+evaluator.getStartupTime());
  }
}

public class Worker extends JSkeleton {
   ...
   public Object run(Object task) {  // this method must be implemented
     Object result;                  // it represents the seq skel body
     ... return(result);             // computes an Object res out of
   }                                 // an Object input task
}
```

Fig. 1. Sample Lithium code: parallel application exploiting task farm parallelism

The compiling tools of the skeleton language or the skeleton libraries take care of automatically deriving/executing actual, efficient parallel code out of the skeleton application without any direct programmer intervention [17, 10].

Lithium, the library we discuss in this work, represents a consistent refinement and development of a former work [7]. The library discussed in [7] just provided the Java programmer with the possibility to implement simple parallel applications exploiting task farm parallelism only. Instead, Lithium:

- provides a reasonable set of fully nestable skeletons, including skeletons that model both data and task parallelism;
- implements the skeletons by fully exploiting a macro data flow execution model [8];
- exploits Java RMI to automatically perform parallel skeleton code execution;
- exploits basic Java reflection features to simplify the skeleton API provided to the programmer;
- allows parallel skeleton programs to be executed sequentially on a single machine, to allow functional debugging to be performed in a simple way.

2 Lithium API

Lithium provides the programmer with a set of (parallel) skeletons that include a `Farm` skeleton, modeling task farm computations[1], a `Pipeline` skeleton, modeling computations structured in independent stages, a `Loop` and a `While` skeleton, modeling determinate and indeterminate iterative computations, an `If` skeleton, modeling conditional computations, a `Map` skeleton, modeling data parallel computations with independent subtasks and a `DivideConquer` skeleton, modeling divide and conquer computations. All the skeletons are provided as subclasses of a `JSkeleton` abstract class.

All skeletons use other skeletons as parameters. As an example, the `Farm` skeleton requires as a parameter another skeleton defining the worker computation, and the `Pipeline` skeleton requires a set of other skeleton parameters, each one defining the computation performed by one of the pipeline stages. Lithium user may encapsulate sequential portions of code in a sequential skeleton by creating a `JSkeleton` subclass[2]. Objects of the subclass can be used as parameters of other, different skeletons.

All the Lithium parallel skeletons implement parallel computation patterns that process a stream of input tasks to compute a stream of output results. As an example, a farm having a worker that computes the function f processes an input task stream with generic element x_i producing the output stream with the corresponding generic element equal to $f(x_i)$, whereas a pipeline with two stages computing function f and g, respectively, processes stream of x_i computing $g(f(x_i))$.

In order to write parallel applications using Lithium skeletons, the programmer should perform the following, (simple) steps:

1. define the skeleton structure of the application. This is accomplished by defining the sequential portions of code used in the skeleton code as `JSkeleton` objects and then using these objects as the parameters of the parallel skeletons (`Pipeline`, `Farm`, etc.) actually used to model the parallel behavior of the application at hand;
2. declare an *evaluator* object (a **Ske** object) and define the program, i.e. the skeleton code defined in the previous step, to be executed by the evaluator as well as the list of hosts to be used to run the parallel code;
3. setup a task pool hosting the initial tasks, i.e. a data structure storing the data items belonging to the input stream to be processed by the program;
4. start the parallel computation, by just issuing an evaluator `parDo()` method call;
5. retrieve the final results, i.e. the stream of output data computed by the program, from a result pool (again, issuing a proper evaluator method call).

Figure 1 outlines the code needed to setup a task farm parallel application processing a stream of input tasks by computing, on each task, the sequential

[1] also known as "embarrassingly parallel" computations

[2] a `JSkeleton` object is an object having a `Object run(Object)` method that represents the sequential skeleton body

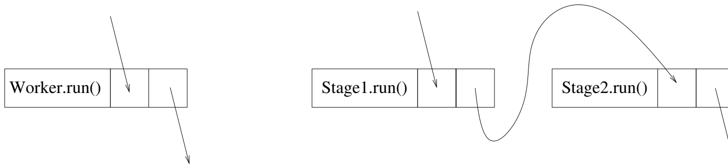

Fig. 2. Macro data flow graphs related to program of Figure 1

code defined in the `Worker` `run` method. The application runs on three processors (the `hosts` ones). The programmer is not required to write any (remote) process setup code, nor any communication, synchronization and scheduling code. He simply issues an `evaluator.parDo()` call and the library automatically computes the `evaluator` program in parallel by forking suitable remote computations on the remote nodes. In case the user simply wants to execute the application sequentially (i.e. to functionally debug the sequential code), he can avoid to issue all the `Ske` evaluator calls. After the calls needed to build the `JSkeleton` program he can simply issue a `run()` method call on the `JSkeleton` object. In that case, the Lithium support performs a completely sequential computation returning the results that the parallel application would return.

All the skeletons defined in Lithium can be defined and used with API calls similar to the ones shown in the Figure (see [21] or look at the source code available at [22]). We want to point out that a very small effort is needed to change the parallel structure of the application, provided that the suitable sequential portions of code needed to instantiate the skeletons are available. In case we understand that the computation performed by the farm workers of Figure 1 can be better expressed with a pipeline of two sequential stages (as an example), we can simply substitute the lines `Worker w = new Worker();` and `Farm f = new Farm(w);` with the lines:

```
Stage1 s1 = new Stage1();    // first seq stage
Stage2 s2 = new Stage2();    // second seq stage
Pipeline p = new Pipeline(); // create the pipeline
p.addWorker(s1);             // setup first pipeline stage
p.addWorker(s2);             // setup second pipeline stage
Farm f = new Farm(p); // create a farm with pipeline workers
```

and we get a perfectly running parallel program computing the results according to a farm of pipeline parallelism exploitation pattern.

3 Lithium implementation

Lithium exploits a macro data flow (MDF, for short) implementation schema for skeletons. The skeleton program is processed to obtain a MDF graph. MDF instructions (MDFi) in the graph represent sequential `JSkeleton` `run` methods.

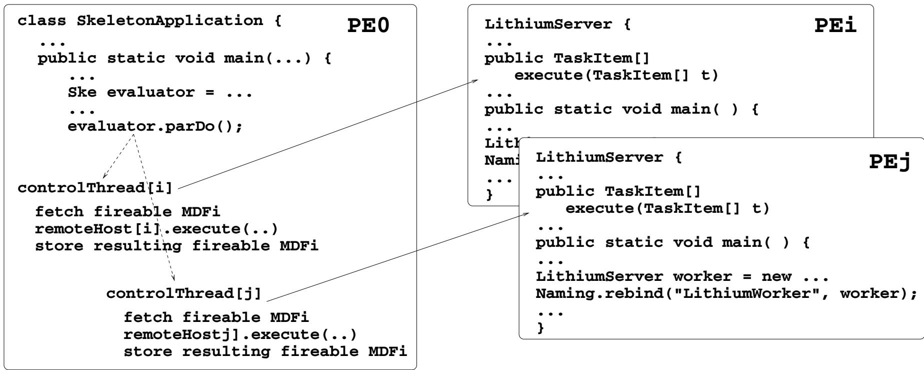

Fig. 3. Lithium architecture

The data flow (i.e. the arcs of MDF graph) is derived by looking at the skeleton nesting structure [6, 8]. The resulting MDF graphs have a single MDFi getting input task (tokens) from the input stream and a single MDFi delivering data items (tokens) to the output stream. As an example, from the application of Figure 1 we derive the MDF graphs of Figure 2: the left one is the one derived from the original application, the right one is the one relative to application with pipelined workers.

The skeleton program is then executed by setting up a server process on each one of the processing elements available and a task pool manager on the local machine. The remote servers are able to compute any one of the fireable MDFi in the graph. A MDF graph can be sent to the servers in such a way that they get specialized to execute only the MDFi in that graph. The local task pool manager takes care of providing a MDFi repository (the *taskpool*) hosting fireable MDFi relative to the MDF graph at hand, and to feed the remote servers with fireable MDFi to be executed.

Logically, any available input task makes a new MDF graph to be instantiated and stored into the taskpool. Then, the input task is transformed into a data flow "token" and dispatched to the proper instruction (the first one) in the new copy of the MDF graph[3]. The instruction becomes fireable and it can be dispatched to one of the remote servers for execution. The remote server computes the MDFi and delivers the result token to one or more MDFi in the taskpool. Such MDFi may (in turn) become fireable and the process is repeated until some fireable MDFi exists in the task pool. Final MDFi (i.e. those dispatching final result tokens/data to the external world) are detected an removed from the taskpool upon `evaluator.readTaskPool()` calls.

Actually, only fireable MDFi are stored in the taskpool. The remote servers know the executing MDF graph and generate fireable complete MDFi to be stored in the taskpool rather than MDF tokens to be stored in already existing, non fireable, MDFi.

[3] different instances of MDF graph are distinguished by a progressive task identifier

Remote servers are implemented as Java RMI servers. A remote server implements a `LithiumInterface`. The interface defines three methods: a `String getVersion()` method, used to check compatibility between local task pool manager and remote servers[4], a `TaskItem[] execute(TaskItem[] task)` method, actually computing a fireable MDFi, and a `void setRemoteWorker(Vector SkeletonList)` method, used to specialize the remote server with the MDF graph currently being executed[5]. RMI implementation has been claimed to demonstrate poor efficiency in the past [16] but recent improvements in JDK allowed us to achieve good efficiency and absolute performance in the execution of skeleton programs, as shown in Section 4. Remote RMI servers must be set up either by hand (via some `ssh hostname rmiregistry &` plus a `ssh hostname java Server &`) or by using proper Perl scripts provided by the Lithium environment.

In the local task pool manager a thread is forked for each one of the remote servers displaced on the remote hosts. Such thread obtains a local reference to a remote RMI server, first; then issues a `setRemoteWorker` remote method call to communicate to the server the MDF graph currently being executed and eventually enters a loop. In the loop body the thread fetches a fireable instruction from the taskpool[6], asks the remote server to compute the MDFi by issuing a remote `execute` method call and deposits the result in the task pool (see Figure 3).

The MDF graph obtained from the `JSkeleton` object used in the `evaluator.setProgram()` call can be processed unchanged or a set of optimization rules can be used to transform the MDF graph (using the `setOptimizations()` and `resetOptimizations()` methods of the `Ske` evaluator class). Such optimization rules implement the "normal form" concept for skeleton trees and basically substitute skeleton subtrees by skeleton subtrees showing a better performance and efficiency in the target machine resource usage. Previous results demonstrated that full stream parallel skeleton subtrees can be collapsed to a single farm skeleton with a (possibly huge) sequential worker leading to a service time which is equal or even better that the service time of the uncollapsed skeleton tree [1]. While developing Lithium we also demonstrated that 1) data parallel skeletons with stream parallel only subtrees can be collapsed to dataparallel skeletons with fully sequential workers, and 2) that normal form skeleton trees require a number of processing elements which is not greater that the number of processing elements needed to execute the corresponding non normal form [21].

As the skeleton program is provided by the programmer as a single (possibly nested) `JSkeleton` object, Java reflection features are used to derive the MDF graph out of it. In particular, reflection and `instanceOf` operators are used to understand the type of the skeleton (as well as the type of the nested

[4] remote server can be run as daemons, therefore they can survive to changes in the local task pool managers

[5] therefore allowing the server to be run as daemon, serving the execution of different programs at different times

[6] using proper `TaskPool synchronized` methods

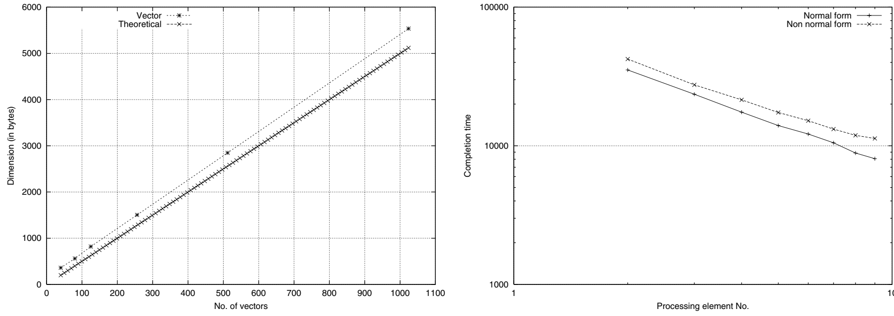

Fig. 4. Serialization overhead (left) and Normal vs. non normal form (right)

skeletons). Furthermore, an `Object[] getSkeletonInfo private` method of the `JSkeleton` abstract class is used to gather the skeleton parameters (e.g. its "body" skeleton). Such method is implemented as a simple `return(null)` statement in the `JSkeleton` abstract class and it is overwritten by each subclass (i.e. by the classes `Farm`, `Pipeline`, etc.) in such a way that it returns in an `Object` vector all the relevant skeleton parameters. These parameters can therefore be inspected by the code building the MDF graph. Without reflection much more info must be supplied by the programmer when defining skeleton nestings in the application code [10].

4 Experiments

We evaluated Lithium performance by performing a full set of experiments on a Beowulf class Linux cluster operated at our Department[7]. The cluster hosts 17 nodes: one node devoted to cluster administration, code development and user interface, and 16 nodes (10 266Mhz Pentium II and 6 400Mhz Celeron nodes) exclusively devoted to parallel program execution. The nodes are interconnected by a (private, dedicated) switched Fast Ethernet network. All the experiments have been performed using Blackdown JDK ports version 1.2.2 and 1.3.

We start considering the overhead introduced by serialization. As data flow tokens happen to be serialized in order to be dispatched to remote executor processes, and as we use Java `Vector` objects to hold tokens, we measured the size overhead of the `Vector` class. Figure 4 (left) reports the results we got, showing that serialization does not add significant amounts of data to the real user data and therefore serialization does not cause significant additional communication overhead.

We measured the differences in the completion time of different applications executed using normal and non normal form. As expected normal form always

[7] the Backus cluster has been implemented in the framework of the Italian National Research Council Mosaico Project

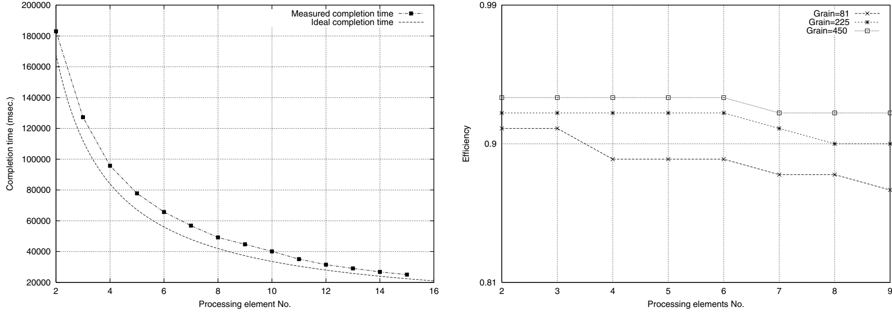

Fig. 5. Ideal vs. measured completion time (left) and Efficiency vs. grain size (right)

performs better that non normal form (see Figure 4 right). The good news are that it performs significantly better and *scales* better (see the behavior when the PE number increases in Fig. 4 right).

Last but not least, we measured the Lithium applications absolute completion time and efficiency related to the computational grain of MDFi. Typical results are drawn in Figure 5. The left plot shows that that Lithium support scales (at least in case of medium to coarse grain computations)[8]. The right plot shows that fairly large grain is required in order to achieve good performance values[9].

All the experiments have been performed using "syntetic" applications. These applications stress different features of the Lithium environment and use skeleton trees (nestings) up to three levels deep, including both task and data parallel skeletons. In addition, we used a couple of simple number crunching applications including Mandelbrot set computation. In all cases the results are similar to the ones presented in the graphs of this Section.

5 Related work

Despite the large number of projects aimed at providing parallel programming environments based on Java, there is no existing project concerning skeletons but the $\mathbf{CO_2P_3S}$ one [14, 13]. Actually this project derives from the design pattern experience [11]. The user is provided with a graphic interface where he can combine different, predefined parallel computation patterns in order to design structured parallel applications that can be run on any parallel/distributed Java platform. In addition, the graphic interface can be used to enter the sequential

[8] the completion times show an additional decrement from 10 nodes on, as the 11th to 16th nodes are more powerful that the first 10 nodes and therefore take a shorter time to execute sequential portions of Java code.

[9] *grain* represents the average computational grain of MDFi. $grain = k$ means that the time spent in the computation of MDFi is k times the time spent in delivering such instructions to the remote servers plus the time spent in gathering results of MDFi execution from the remote servers

portions of Java code needed to complete the patterns. The overall environment is layered in such a way that the user designs the parallel application using the patterns, then those patterns are implemented exploiting a layered implementation framework. The framework gradually exposes features of the implementation code thus allowing the programmer to perform fine performance tuning of the resulting parallel application. The whole object adopt a quite different approach with respect to our one, especially in that it does not use any kind of macro data flow technique in the implementation framework. Instead, parallel patterns are implemented by process network templates directly coded in the implementation framework. However, the final result is basically the same: the user is provided with an high level parallel programming environment that can be used to derive high performance parallel Java code running on parallel/distributed machines.

Macro data flow implementation techniques, instead, have been used to implement skeleton based parallel programming environments by Serot in the Skipper project [19, 18]. Skipper is an environment supporting skeleton based, parallel image processing application development. The techniques used to implement Skipper are derived from the same results we start with to design Lithium, although used within a different programming enviroment.

6 Conclusions and future work

We described a new Java parallel programming environment providing the user with the possibility to model all the parallel behavior of his applications by using predefined skeletons. This work significantly extends [7] as both data and stream parallel skeletons are implemented, and optimisation rules are provided that improve execution efficiency. Being based on skeletons, the Lithium environment relieves the programmer of all the error prone activities related to process setup, mapping and scheduling, communication and synchronization handling, etc. that must usually be dealt with when programming parallel applications. Lithium is the first full fledged, skeleton based parallel programming environment written in Java and implementing skeleton parallel execution by using macro data flow techniques. We performed experiments with Lithium that demonstrate that good scalability and efficiency values can be achieved. Lithium and is currently available as open source at `http://massivejava.sourceforge.net`.

References

1. M. Aldinucci and M. Danelutto. Stream parallel skeleton optimisations. In *Proc. of the IASTED International Conference Parallel and Distributed Computing and Systems*, pages 955–962. IASTED/ACTA Press, November 1999. Boston, USA.
2. G. Antoniu, L. Bougé, P. Hatcher, M. MacBeth, K. McGuigan, and R. Namyst. "Compiling Multithreaded Java Bytecode for Distributed Execution". In A. Bode, T. Ludwig, W. Karl, and R. Wismuller, editors, *"EuroPar 2000 - Parallel Processing"*, number 1900 in LNCS, pages 1039–1052. Springer Verlag, 2000.

3. P. Au, J. Darlington, M. Ghanem, Y. Guo, H.W. To, and J. Yang. Co-ordinating heterogeneous parallel computation. In L. Bouge, P. Fraigniaud, A. Mignotte, and Y. Robert, editors, *Europar '96*, pages 601–614. Springer-Verlag, 1996.
4. B. Bacci, M. Danelutto, S. Pelagatti, and M. Vanneschi. SkIE: a heterogeneous environment for HPC applications. *Parallel Computing*, 25:1827–1852, Dec 1999.
5. M. Cole. *Algorithmic Skeletons: Structured Management of Parallel Computations*. Research Monographs in Parallel and Distributed Computing. Pitman, 1989.
6. M. Danelutto. Dynamic Run Time Support for Skeletons. In E. H. D'Hollander, G. R. Joubert, F. J. Peters, and H. J. Sips, editors, *Proceedings of the International Conference ParCo99*, volume Parallel Computing Fundamentals & Applications, pages 460–467. Imperial College Press, 1999.
7. M. Danelutto. Task farm computations in java. In Buback, Afsarmanesh, Williams, and Hertzberger, editors, *High Performance Computing and Networking*, LNCS, No. 1823, pages 385–394. Springer Verlag, May 2000.
8. M. Danelutto. Efficient support for skeletons on workstation clusters. *Parallel Processing Letters*, 11(1):41–56, 2001.
9. M. Danelutto, R. Di Cosmo, X. Leroy, and S. Pelagatti. Parallel Functional Programming with Skeletons: the OCAMLP3L experiment. In *ACM Sigplan Workshop on ML*, pages 31–39, 1998.
10. M. Danelutto and M. Stigliani. SKElib: parallel programming with skeletons in C. In A. Bode, T. Ludwing, W. Karl, and R. Wismüller, editors, *Euro-Par 2000 Parallel Processing*, LNCS, No. 1900, pages 1175–1184. Springer Verlag, August/September 2000.
11. E. Gamma, R. Helm, R. Johnson, and J. Vissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, 1994.
12. jPVM. "http://www.chmsr.gatech.edu/jPVM/". The jPVM home page, 2001.
13. S. McDonald, D. Szafron, J. Schaeffer, and S. Bromling. From Patterns to Frameworks to Parallel Programs. submitted to Journal of Parallel and Distributed Computing, December 2000.
14. S. McDonald, D. Szafron, J. Schaeffer, and S. Bromling. Generating Parallel Program Frameworks from Parallel Design Patterns. In A. Bode, T. Ludwing, W. Karl, and R. Wismüller, editors, *Euro-Par 2000 Parallel Processing*, LNCS, No. 1900, pages 95–105. Springer Verlag, August/September 2000.
15. MpiJava. "http://www.npac.syr.edu/projects/pcrc/mpiJava/". The MpiJava home page, 2001.
16. C. Nester, R. Philippsen, and B. Haumacher. "A More Efficient RMI for Java". In *ACM 1999 Java Grande Conference*, pages 152–157, June 1999.
17. S. Pelagatti. *Structured Development of Parallel Programs*. Taylor & Francis, 1998.
18. J. Serot. "Putting skeletons at work. An overview of the SKIPPER project". PARCO'2001 workshop on *Advanced Environments for Parallel and Distributed Computing*, to appear, September 2001.
19. J. Serot, D. Ginhac, R. Chapuis, and J. Derutin. "Fast prototyping of parallel-vision applications using functional skeletons". *Machine Vision and Applications*, 12:217–290, 2001. Springer Verlag.
20. Sun. "The Java home page". `http://java.sun.com`, 2001.
21. P. Teti. "Lithium: a Java skeleton environment". (*in italian*) Master's thesis, Dept. Computer Science, University of Pisa, October 2001.
22. P. Teti. "`http://massivejava.sourceforge.net`". home page of the Lithium project at `sourceforge.net`, 2001.