Available at
www.ComputerScienceWeb.com
POWERED BY SCIENCE DIRECT®

Future Generation Computer Systems 19 (2003) 611–626

FGCS
FUTURE GENERATION COMPUTER SYSTEMS
www.elsevier.com/locate/future

An advanced environment supporting structured parallel programming in Java

M. Aldinucci, M. Danelutto*, P. Teti

Department of Computer Science, University of Pisa, V. Buonarroti 2, 56127 Pisa, Italy

Abstract

In this work we present Lithium, a *pure Java* structured parallel programming environment based on skeletons (common, reusable and efficient parallelism exploitation patterns). Lithium is implemented as a Java package and represents both the first skeleton based programming environment in Java and the first complete skeleton based Java environment exploiting macro-data flow implementation techniques.

Lithium supports a set of user code optimizations which are based on skeleton rewriting techniques. These optimizations improve both absolute performance and resource usage with respect to original user code. Parallel programs developed using the library run on any network of workstations provided the workstations support plain JRE. The paper describes the library implementation, outlines the optimization techniques used and eventually presents the performance results obtained on both synthetic and real applications.
© 2002 Elsevier Science B.V. All rights reserved.

Keywords: Java; Parallel programming; Skeletons; Macro-data flow; Optimizations

1. Introduction

The development of efficient parallel programs is really a hard task. Besides coding all the algorithm details, the programmer must also take care of the details involved in parallelism exploitation: concurrent activity setup (either processes or threads), mapping and scheduling, communication and synchronization handling, data allocation, etc. These activities are usually fully in charge of the programmer and constitute a difficult, error prone programming effort. The effort required to the programmer vary from moderate to high, depending on the programming language/environment chosen to develop parallel applications.

The Java programming environment includes features that can be naturally used to address network and distributed computing: JVM and bytecode, multi-threading, remote method invocation, socket and security handling and, more recently, JINI, Java Spaces, Servlets, etc. [1]. Many parallel/distributed applications have been developed using these features [2]. Also, many efforts have been performed to make Java a more suitable programming environment for parallel computing. In particular, 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 [3,4]. 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. As an example extensions of the JVM have been designed that allow

* Corresponding author.
E-mail address: marcod@di.unipi.it (M. Danelutto).

plain Java threads to be run in a seamless way on the different processors of a single SMP machine [5,6]. In the latter, libraries are supplied to the programmer that simply uses them within his parallel code [7,8].

In this work we discuss a new Java parallel programming environment, Lithium, which differs from the environments cited above and which is meant to be a further step in the direction of the design of user friendly, efficient, parallel programming environments based on Java. Lithium is a Java library that supports *structured* parallel programming. It is based on the *algorithmical skeleton* concept. Skeletons have been originally conceived by Cole [9] and then used by different research groups to design high performance structured parallel programming environments [10–12]. A skeleton is basically an abstraction modeling a common, reusable parallelism exploitation pattern. Skeletons can be provided to the programmer either as language constructs [10,11] or as libraries [13–15]. They can be nested to structure complex parallel applications. 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 [14,16].

In order to write a working parallel application using a skeleton based parallel programming environment, the programmer must usually perform the following steps:

- first, the programmer must express the *parallel structure* of the application by using a proper skeleton nesting;
- then, the programmer must write the *application specific sequential portions of code* used as skeleton parameters;
- eventually the programmer must simply *compile and link* the resulting code to obtain running, parallel object code.

Lithium provides the programmer with a full Java, skeleton based parallel programming environment. The library supports common skeletons, including pipelines, task farms, iterative and data parallel skeletons. Using Lithium, the programmer can setup a parallel application instantiating the skeletons provided, nesting them, providing tasks to be computed (input data), asking the parallel computation of the resulting program on a set of interconnected workstations and eventually get (and use) the results of such parallel computation.

As an example, the programmer can express his parallel application as a pipeline having stages that are either sequential or exploit data parallelism. Then, he can prepare the sequential portions of code implementing sequential stages and those implementing data decomposition, processing and recomposition relative to the data parallel stages. He can denote such portions of code as the proper pipeline, data parallel stages. Eventually he can compile and run the resulting program. When a working instance of the application has been obtained and tested, the programmer can start a *performance refinement* step. During this activity he can tune either the skeleton structure or the skeleton parameters in such a way that performance bottlenecks are removed/mitigated [16].

The implementation of Lithium fully exploits Java RMI to distribute computations across different processing elements of the target architecture. Java reflection features are also exploited to make the Lithium API simpler. Last but not least, the clean OO structure of Lithium code also allows the sequential execution (emulation, actually) of a parallel program onto a single machine. This is a very useful feature during functional application code debugging. Actually, Lithium represents a consistent refinement and development of a former work [17]. Main differences lay in the larger skeleton set implemented ([17] only handles embarrassingly parallel computations, while Lithium provides a complete skeleton set) and in the implementation of a set of optimization rules that may significantly enhance skeleton program performances. The optimization rules implemented in Lithium extend the ones discussed in [18,19]. They have not yet been used in any other skeleton based programming environment.

This paper is organized as follows. Section 2 describes the skeletons provided by Lithium, Section 3 describes the optimization strategies implemented by Lithium (and available on user request), Section 4 outlines the Lithium API. Section 5 describes how the library is implemented exploiting macro-data flow, Section 6 reports the experimental results achieved and Section 7 deals with related work.

2. Lithium skeletons

Lithium provides both task parallel and data parallel skeletons. All the skeletons work on streams, i.e. process a stream of input tasks to produce a stream of results. We denote streams[1] by angled braces (e.g. $\langle x_1, x_2, \ldots, \tau\rangle$) and we use a non-strict stream constructor denoted by :: having type (Stream $\times$ Stream $\to$ Stream). Empty streams are denoted by $\langle\,\rangle$.

All the skeletons are assumed to be stateless. No concept of "global state" is supported by the implementation, but the ones explicitly programmed by the user, possibly exploiting plain Java mechanisms (e.g. RMI servers encapsulating shared data structures, whose services/methods are invoked by code used to express computations within skeletons). In particular, static (class) variables cannot be used in the definition of Lithium code to share information across different concurrent/parallel entities.

The Lithium skeletons are fully nestable. Each skeleton has skeleton type parameters that model the computations encapsulated in the related parallelism exploitation pattern.

The skeletons (Δ) provided by Lithium are defined as follows:

$$\begin{aligned}\Delta ::= \;& \mathsf{seq}\; f \mid \\ & \mathsf{farm}\; \Delta \mid \mathsf{pipe}\; \Delta_1 \Delta_2 \mid \mathsf{comp}\; \Delta_1 \Delta_2 \mid \\ & \mathsf{map}\; f_{\mathsf{d}}\; \Delta\; f_{\mathsf{c}} \mid \mathsf{d\&c}\; f_{\mathsf{tc}}\; f_{\mathsf{d}}\; \Delta\; f_{\mathsf{c}} \mid \\ & \mathsf{for}\; i\, \Delta \mid \mathsf{while}\; f_{\mathsf{tc}}\; \Delta \mid \mathsf{if}\; f_{\mathsf{tc}}\; \Delta_1 \Delta_2\end{aligned}$$

$f, f_{\mathsf{c}}, f_{\mathsf{d}}, f_{\mathsf{tc}} ::=$ *Sequential Java functions*.

and a Lithium program is a skeleton expression:

$$\mathit{Lithium\_prog} ::= \Delta : \langle\sigma\rangle \to \langle\tau\rangle,$$

processing a stream of input data $\langle\sigma\rangle$ and producing a stream of output results $\langle\tau\rangle$.

Sequential Java functions[2] f_{c}, f_{d} represent families of functions that enable the splitting of a singleton stream in tuples of singleton streams and vice versa: $f_{\mathsf{c}} : \langle\circ\rangle^* \to \langle\circ\rangle$ and $f_{\mathsf{d}} : \langle\circ\rangle \to \langle\circ\rangle^*$, being $\langle\circ\rangle$ the singleton stream. As an example f_{c}, f_{d} may be used to split an array in their rows/columns and join them back to the original array shape.

[1] Both finite and infinite streams, i.e. lists and non-strict lists, respectively.

[2] Namely the `run()` method call of an instance of `JSkeletons` class (see Section 4).

Intuitively, seq skeleton just encapsulates sequential portions of code in such a way they can be used as parameters of other skeletons; farm and pipe skeletons model usual task farm (*alias* embarrassingly parallel) computations and computations organized in *stages*; comp models pipelines with stages executed serially (on the same processing element); map models data parallel computations: f_{d} decomposes the input data into a set of possibly overlapping data subsets, the inner skeleton computes a result out of each subset and the f_{c} function rebuilds a unique result out of these results; d&c models divide and conquer computations: input data is divided into subsets by f_{d} and each subset is computed recursively and concurrently until the f_{tc} condition does not hold true. At this point results of subcomputations are conquered via the f_{c} function. Last but not least, for, while and if skeletons model finite iteration, indefinite iteration and conditional.

More formally, the operational semantics of the Lithium skeletons is described in Fig. 1 (a full version of these semantics appears in [20]. Here curly braces denote tuples (e.g. $\{x_1, \ldots, x_k\}$), α represents the apply-to-all on tuples ($\alpha\Delta\{x_1, \ldots, x_k\} = \{\Delta x_1, \ldots, \Delta x_k\}$) and all the functions used are assumed to be strict, but the infix stream constructor :: which is only strict in his first argument[3] and the apply-to-all function α.

Actually, in all those cases where a rule such as

$$\mathsf{Skel}\ldots\langle x, \tau\rangle_{\ell_1} \to \mathcal{F}(\langle x\rangle_{\ell_2}) :: \mathsf{Skel}\ldots\langle\tau\rangle_{\ell_3},$$

holds,[4] then two further rules hold, which is not summarized in Fig. 1 to avoid clobbering the figure:

$$\mathsf{Skel}\ldots\langle x\rangle_{\ell_1} \to \mathcal{F}(\langle x\rangle_{\ell_2}) \quad \text{and} \quad \mathsf{Skel}\ldots\langle\,\rangle \to \langle\,\rangle.$$

In the computation of a Lithium skeleton program parallelism comes from two distinct sources:

- *data parallelism*: all the computations in the α apply-to-all may be performed in parallel (in map and d&c skeletons);
- *task parallelism*: each item of a stream (e.g. a (strict) function application appearing within some :: operators) may be computed in parallel with any other function application appearing in the same stream provided that their labels differ.

[3] Therefore :: evaluates arguments left-to-right.

[4] $\mathsf{Skel} \in$ [seq, farm, pipe, comp, map, d&c, for, while, if].

1. $\mathsf{seq}\ f\ \langle x,\tau\rangle_\ell \to \langle f\,x\rangle_\ell :: \mathsf{seq}\ f\ \langle\tau\rangle_\ell$
2. $\mathsf{farm}\ \Delta\ \langle x,\tau\rangle_\ell \to \Delta\ \langle x\rangle_{\mathcal{O}(\ell,x)} :: \mathsf{farm}\ \Delta\ \langle\tau\rangle_{\mathcal{O}(\ell,x)}$
3. $\mathsf{pipe}\ \Delta_1\ \Delta_2\ \langle x,\tau\rangle_\ell \to \Delta_2\ [\Delta_1\ \langle x\rangle_\ell]_{\mathcal{O}(\ell,x)} :: \mathsf{pipe}\ \Delta_1\ \Delta_2\ \langle\tau\rangle_\ell$
4. $\mathsf{comp}\ \Delta_1\ \Delta_2\ \langle x,\tau\rangle_\ell \to \Delta_2\ [\Delta_1\ \langle x\rangle_\ell]_\ell :: \mathsf{comp}\ \Delta_1\ \Delta_2\ \langle\tau\rangle_\ell$
5. $\mathsf{map}\ f_c\ \Delta\ f_d\ \langle x,\tau\rangle_\ell \to f_c\ (\alpha\ \Delta)\ f_d\ \langle x\rangle_\ell :: \mathsf{map}\ f_c\ \Delta\ f_d\ \langle\tau\rangle_\ell$
6. $\mathsf{d\&c}\ f_{tc}\ f_c\ \Delta\ f_d\ \langle x,\tau\rangle_\ell \to \mathsf{d\&c}\ f_{tc}\ f_c\ \Delta\ f_d\ \langle x\rangle_\ell :: \mathsf{d\&c}\ f_{tc}\ f_c\ \Delta\ f_d\ \langle\tau\rangle_\ell$
$$\mathsf{d\&c}\ f_{tc}\ f_c\ \Delta\ f_d\ \langle y\rangle_\ell = \begin{cases} \Delta\ \langle y\rangle_\ell & \textit{iff}\ \ (f_{tc}\ y) \\ f_c\,(\alpha\,(\mathsf{d\&c}\ f_{tc}\ f_c\ \Delta\ f_d))\ f_d\ \langle y\rangle_\ell & \textit{otherwise} \end{cases}$$
7. $\mathsf{for}\ i\ \Delta\ \langle x,\tau\rangle_\ell \to \underbrace{\Delta(\Delta(\cdots(\Delta}_{i\ \text{times}}\langle x\rangle_\ell)\cdots)) :: \mathsf{for}\ i\ \Delta\ \langle\tau\rangle_\ell$
8. $\mathsf{while}\ f_{tc}\ \Delta\ \langle x,\tau\rangle_\ell \to \begin{cases} \mathsf{while}\ f_{tc}\ \Delta\ \ (\Delta\ \langle x\rangle_\ell :: \langle\tau\rangle_\ell) & \textit{iff}\ \ (f_{tc}\ x) \\ \langle x\rangle_\ell ::\ \mathsf{while}\ f_{tc}\ \Delta\ \langle\tau\rangle_\ell & \textit{otherwise} \end{cases}$
9. $\mathsf{if}\ f_{tc}\ \Delta_1\ \Delta_2\ \langle x,\tau\rangle_\ell \to \begin{cases} \Delta_1\langle x\rangle_\ell :: \mathsf{if}\ f_{tc}\ \Delta_1\ \Delta_2\ \langle\tau\rangle_\ell & \textit{iff}\ \ (f_{tc}\ x) \\ \Delta_2\langle x\rangle_\ell :: \mathsf{if}\ f_{tc}\ \Delta_1\ \Delta_2\ \langle\tau\rangle_\ell & \textit{otherwise} \end{cases}$

Fig. 1. Lithium operational semantic. $x,\ y \in$ Value; $\sigma, \tau \in$ Value*; $\ell, \ell_i, \ldots, \in$ Label = Strings $\cup\ \{\bot\}$; $\mathcal{O}$: Label $\times$ Value $\to$ Label.

In fact, in this operational semantics, streams are labeled and different skeletons behave differently with respect to labels. The idea here is that labels are used to distinguish computations that may be performed in parallel from those that may not. Function $\mathcal{O}(\ell, x)$ simply returns a "fresh", unused label built using information coming from the previously used label ℓ and from the current data item x.

Therefore, each data item processed by a farm is given a fresh label, modeling the fact that embarrassingly parallel task farm computations can all possibly happen concurrently. Pipeline keeps labels in such a way that first and second stage cannot be computed in parallel on the same data item.

As an example, consider the computations described in Fig. 2. The operational semantics rewrites the farm computation relative to a six item stream by

$\mathsf{farm}\ (\mathsf{seq}\ f)\ \langle x_1,x_2,x_3,x_4,x_5,x_6\rangle_\bot \to^*$
$\quad \mathsf{seq}\ f\ \langle x_1\rangle_1 :: \mathsf{seq}\ f\ \langle x_2\rangle_2 :: \mathsf{seq}\ f\ \langle x_3\rangle_3 :: \mathsf{farm}\ (\mathsf{seq}\ f\ )\ \langle x_4,x_5,x_6\rangle_\bot \to^*$
$\quad \mathsf{seq}\ f\ \langle x_1\rangle_1 :: \mathsf{seq}\ f\ \langle x_2\rangle_2 :: \mathsf{seq}\ f\ \langle x_3\rangle_3 :: \mathsf{seq}\ f\ \langle x_4\rangle_4 :: \mathsf{seq}\ f\ \langle x_5\rangle_5 :: \mathsf{seq}\ f\ \langle x_6\rangle_6 \to^*$
$\quad \langle f\,x_1\rangle_1 :: \langle f\,x_2\rangle_2 :: \langle f\,x_3\rangle_3 :: \langle f\,x_4\rangle_4 :: \langle f\,x_5\rangle_5 :: \langle f\,x_6\rangle_6 \to^*$
$\quad \langle f\,x_1,\ f\,x_2,\ f\,x_3,\ f\,x_4,\ f\,x_5,\ f\,x_6\rangle_\bot$

$\mathsf{pipe}\ (\mathsf{seq}\ f)\ (\mathsf{seq}\ g)\langle x_1,x_2,x_3,x_4\rangle_\bot \to^*$
$\quad (\mathsf{seq}\ g)\,[\mathsf{seq}\ f\ \langle x_1\rangle_\bot]_1 :: (\mathsf{seq}\ g)\,[\mathsf{seq}\ f\ \langle x_2\rangle_\bot]_1 :: \mathsf{pipe}\ (\mathsf{seq}\ f)\ (\mathsf{seq}\ g)\langle x_3,x_4\rangle_\bot \to^*$
$\quad (\mathsf{seq}\ g)\,[\mathsf{seq}\ f\ \langle x_1\rangle_\bot]_1 :: (\mathsf{seq}\ g)\,[\mathsf{seq}\ f\ \langle x_2\rangle_\bot]_1 :: (\mathsf{seq}\ g)\,[\mathsf{seq}\ f\ \langle x_3\rangle_\bot]_1 :: (\mathsf{seq}\ g)\,[\mathsf{seq}\ f\ \langle x_4\rangle_\bot]_1 \to^*$
$\quad \mathsf{seq}\ g\ \langle f\ x_1\rangle_1 :: (\mathsf{seq}\ g)\,[\mathsf{seq}\ f\ \langle x_2\rangle_\bot]_1 :: (\mathsf{seq}\ g)\,[\mathsf{seq}\ f\ \langle x_3\rangle_\bot]_1 :: (\mathsf{seq}\ g)\,[\mathsf{seq}\ f\ \langle x_4\rangle_\bot]_1 \to^*$
$\quad \langle g\circ f\ x_1\rangle_1 :: \mathsf{seq}\ g\ \langle f\ x_2\rangle_1 :: (\mathsf{seq}\ g)\,[\mathsf{seq}\ f\ \langle x_3\rangle_\bot]_1 :: (\mathsf{seq}\ g)\,[\mathsf{seq}\ f\ \langle x_4\rangle_\bot]_1 \to^*$
$\quad \langle g\circ f\ x_1\rangle_1 :: \langle g\circ f\ x_2\rangle_1 :: \mathsf{seq}\ g\ \langle f\ x_3\rangle_1 :: (\mathsf{seq}\ g)\,[\mathsf{seq}\ f\ \langle x_4\rangle_\bot]_1 \to^*$
$\quad \langle g\circ f\ x_1\rangle_1 :: \langle g\circ f\ x_2\rangle_1 :: \langle g\circ f\ x_3\rangle_1 :: \langle g\circ f\ x_4\rangle_1 \to^*$
$\quad \langle g\circ f\ x_1,\ g\circ f\ x_2,\ g\circ f\ x_3,\ g\circ f\ x_4\rangle_\bot$

Fig. 2. Stream label usage examples.

an expression involving seq skeletons (first half of Fig. 2). All the seq skeletons work on singleton stream with *different* labels. Therefore all the $\mathsf{seq}\, f \langle x_i \rangle_{l_j}$ can be possibly computed concurrently. When pipelines are used (second half of Fig. 2) another seq based expression is derived, whose terms happen to have all the same label. Therefore, at the beginning only one seq skeleton can be computed, i.e. the rightmost of those with the $\perp$ label. From this step on, two expressions happen to have a different label: one relative to the computation of the first pipeline stage (with $\perp$ label) and one relative to the computation of the second pipeline stage (with one label): these skeleton expressions can actually be computed in parallel.

Lithium supports most of the skeletons discussed in previous skeleton/structured parallel programming works [10–12,16,21], and allows a reasonably large number of parallel applications to be implemented. The architectural design of Lithium also allows relatively easy extension of the skeleton set (see Section 5) in case it is needed.

3. Skeleton optimizations

We define a *stream parallel skeleton composition* as a skeleton expression only holding pipe, farm and seq skeletons. For such composition we inductively define the *fringe* (ϕ) as follows:

$$\phi(\Delta) = \begin{cases} \mathsf{seq}\, f & \Delta = \mathsf{seq}\, f \\ \mathsf{comp}\, \phi(\Delta_1)\, \phi(\Delta_2) & \Delta = \mathsf{pipe}\, \Delta_1 \Delta_2 \\ \phi(\Delta_w) & \Delta = \mathsf{farm}\, \Delta_w \end{cases}$$

In [18] we demonstrated that for any stream parallel skeleton composition Δ a *normal form* $\bar{\Delta}$ exists (being $\bar{\Delta} = \mathsf{farm}\, \phi(\Delta)$) such that it computes the same program computed by Δ with a performance equal or better than the one of the original skeleton composition Δ (i.e. $T_s(\bar{\Delta}) \leq T_s(\Delta)$, being $T_s(\Delta)$ the service time of the skeleton program Δ). Equivalence of normal and non-normal form is derived by using the skeleton “functional” semantics that can be easily derived from the operational semantics described in Fig. 1. Relationships between performance of normal and non-normal form are derived using a simple, ideal, log P-like performance model taking into account both sequential computation time and communication time. As an example, the performance model defines $T_s(\mathsf{farm}\, \Delta)$ as the $\min\{\max\{T_i(\Delta), T_o(\Delta)\}, T_s(\Delta)\}$. Here, T_i and T_o represent the time spent in delivering a new task and retrieving the computed result to and from the processing element computing the task, respectively. In other words, the time needed to accept a new input task in an embarrassingly parallel computation (the stateless farm) is determined by the minimum between the time spent in communicating data to and from the remote processing elements and the time spent in actually computing results out of the input tasks.

Starting from these results, while developing Lithium we also demonstrated two further results concerning skeleton tree (nesting) optimizations. The first one extends normal form to data parallel skeleton nesting with stream parallel only workers:

> *given a skeleton program* $\Delta = \mathsf{map}\, f_{\mathsf{d}} \Delta_w f_{\mathsf{c}}$ *with* Δ_w *being a stream parallel skeleton composition, a normal form exists* $\bar{\Delta} = \mathsf{map}\, f_{\mathsf{d}} \overline{\Delta_w} f_{\mathsf{c}}$ *such that* $T_s(\bar{\Delta}) \leq T_s(\Delta)$.

The second one concerns resources (processing elements) used by normal and non-normal form programs. In theory, and according to the operational semantics of Fig. 1, the execution of a skeleton program needs, at any time, a set of processing elements holding a distinct processing element for each concurrent activity.[5] The amount of resources needed to compute a skeleton program is the maximum number of elements in this set measured during the whole program execution. We denote such number by $\#(\Delta)$. Under this assumptions:

> *for any skeleton program* Δ *being either a stream parallel skeleton composition or a* $\mathsf{map}\, f_{\mathsf{d}} \Delta_w f_{\mathsf{c}}$ *with* Δ_w *a stream parallel skeleton composition then* $\#(\bar{\Delta}) \leq \#(\Delta)$.

A full proof of these new results is described in [22]. The point we want to make here is that these results guarantee that Lithium can perform effective optimizations in the execution of skeleton code. Actually, Lithium performs automatical transformation of skeleton nestings into normal form, before proceeding to compute the programs. The user may explicitly ask to avoid such transformations.

[5] That is, each one of the function applications labeled with different labels and each one of the computations happening within an apply-to-all.

4. Lithium API

Lithium provides the programmer with a set of classes implementing the skeletons described in Section 2. The classes can be used to instantiate objects that will populate the skeleton nesting modeling the parallel behavior of the applications. All the skeletons are provided as subclasses of a `JSkeletons` abstract class. This class defines two abstract methods: the first one is a `public Object run(Object)` method. It is used to encapsulate a sequential portion of code (in case of sequential skeleton) or the code that sequentially emulates the parallel skeleton behavior (in case of the other, non-sequential skeletons). The second method defined is a protected `Object[ ] getSkeletonInfo( )` method, which is basically used by Lithium to gather the information needed to build the application skeleton nesting.

Therefore, a Lithium sequential skeleton is nothing but a subclass of the Lithium abstract class `JSkeletons` providing an implementation of the abstract method `public Object run(Object)`, while a farm is modeled via the `Farm JSkeletons` subclass, the pipe via the `Pipe` one, etc. All the details relative to skeleton definition in Lithium can be found in [22] as well as at http://massivejava.sourceforge.net, where the whole code is available as open source (including its Javadoc documentation).

Beside defining skeleton nestings, Lithium API provides a way to execute such skeleton programs. This is accomplished through objects of the `Ske` class. This class provides the object actually taking a skeleton program, a set of input tasks and providing to compute the program in parallel. After creating a `Ske` object, a `setProgram(JSkeletons pgm)` method can be invoked to define which skeleton program is to be executed, an `addHosts(String [ ] hostlist)` method can be called to provide the names of the machines to be used for parallel execution, some `setupTaskPool(Object task)` can be invoked to provide the task items of the input stream, a `parDo( )` method call can be issued to start parallel program execution and eventually `Object readTaskPool( )` method can be invoked to read the results computed.

In summary, in order to write parallel applications using Lithium skeletons, the programmer should perform the following steps:

1. define the skeleton structure of the application;
2. declare a **Ske** object and define the program (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;
4. start the parallel computation issuing a `parDo( )` method call;
5. retrieve and process the final results.

Fig. 3 outlines the code needed to setup a task farm parallel application processing a stream of input tasks by computing, on each task, the sequential code defined in the `Worker run` method. The application runs on three processors (the `hosts` ones). The programmer is neither required to write any (remote) process setup code, nor any communication, synchronization and scheduling code. He 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

```
import lithium.*;  ...
public class SkeletonApplication {
  public static void main(String [] args) {
    ...
    // define skel. program
    Worker w = new Worker();
    Farm f = new Farm(w);
    // setup evaluator
    Ske evaluator = new Ske();
    evaluator.setProgram(f);
    String [] hosts = {"alpha1","alpha2",
                       "131.119.5.91"};
    evaluator.addHosts(hosts);
    // prepare input tasks
    for(int i=0;i<ntasks;i++)
      evaluator.setupTaskPool(task[i]);
    evaluator.stopStream();
    evaluator.parDo();
    // ask parallel computation
    while(!evaluator.isResEmpty()) {
        Object res = evaluator.readTaskPool();
        ...      // consume results
     }
  }}
```

Fig. 3. Sample Lithium code: parallel application exploiting task farm parallelism.

debug the sequential code), he can avoid to issue all the `Ske` evaluator calls. After the calls needed to build the `JSkeletons` program he can issue a `run( )` method call on the `JSkeletons` object. In that case, the Lithium support performs a completely sequential computation returning the results that the parallel application would eventually return. We want to point out that, in case we understand that the computation performed by the farm workers (Fig. 3) can be better expressed by a functional composition, we can arrange things in such a way that farm workers are two stage pipelines. This can be achieved by substituting the lines `Worker w = new Worker( );` and `Farm f=new Farm(w);` with the lines:

```
Stage1 s1 = new Stage1( );
Stage2 s2 = new Stage2( );
Pipeline p = new Pipeline( );
p.addWorker(s1);
p.addWorker(s2);
Farm f = new Farm(p);
```

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

5. Lithium implementation

Lithium exploits a macro-data flow (MDF) implementation schema for skeletons. The skeleton program is processed to obtain a MDF graph. MDF instructions (MDFi) in the graph represent sequential `JSkeletons run` methods. Data flow (i.e. the arcs of MDF graph) is derived by looking at the skeleton nesting structure [23,24]. 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.

The skeleton program is executed in Lithium 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.

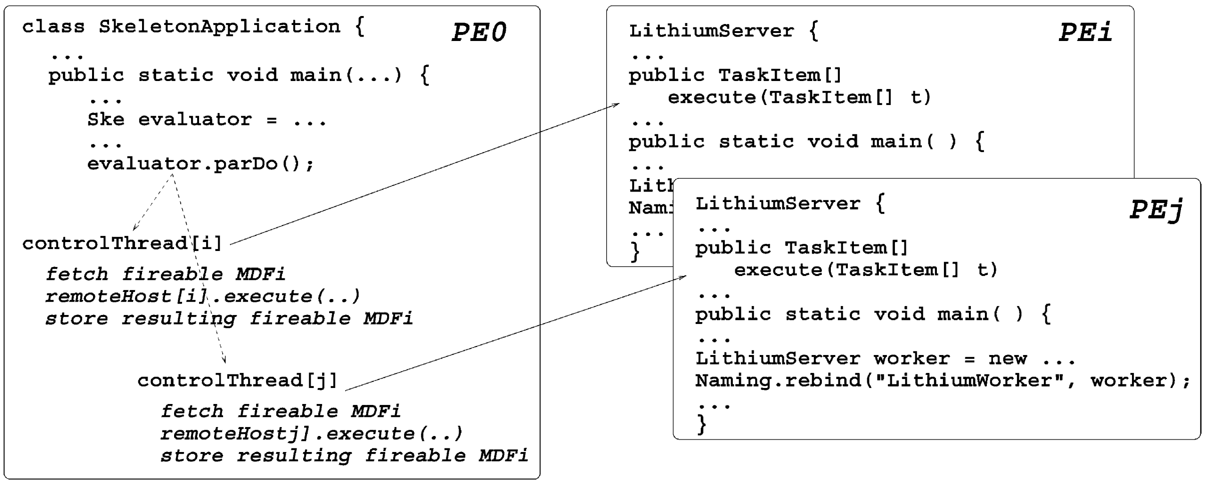
Logically, any available input task makes a new MDF graph to be instantiated and stored into the taskpool. Then, the input task is transformed into a data flow “token” and dispatched to the proper instruction (the first one) in the new copy of the MDF graph.[6] The instruction becomes fireable and it 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 and 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.

Remote servers are implemented as Java RMI servers. A remote server implements a `LithiumInterface`. This interface defines two main methods: 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.[7] RMI implementation has been claimed to demonstrate poor efficiency in the past [25] but recent improvements in JDK allowed us to achieve good efficiency and absolute performance in the execution of skeleton programs, as shown in Section 6. Remote RMI servers must be setup either by hand (via some `ssh hostname Java Server &` command) or by using proper Perl scripts provided by the Lithium environment.

[6] Different instances of MDF graph are distinguished by a progressive task identifier.

[7] Therefore allowing the server to be run as daemon, serving the execution of different programs at different times.



```
class SkeletonApplication {
   ...
   public static void main(...) {
      ...
      Ske evaluator = ...
      ...
      evaluator.parDo();

controlThread[i]
  fetch fireable MDFi
  remoteHost[i].execute(..)
  store resulting fireable MDFi

        controlThread[j]
          fetch fireable MDFi
          remoteHostj].execute(..)
          store resulting fireable MDFi
```

```
LithiumServer {
...
public TaskItem[]
   execute(TaskItem[] t)
...
public static void main( ) {
...
Lit
Nam
...
}
```

```
LithiumServer {
...
public TaskItem[]
   execute(TaskItem[] t)
...
public static void main( ) {
...
LithiumServer worker = new ...
Naming.rebind("LithiumWorker", worker);
...
}
```

Fig. 4. Lithium architecture.

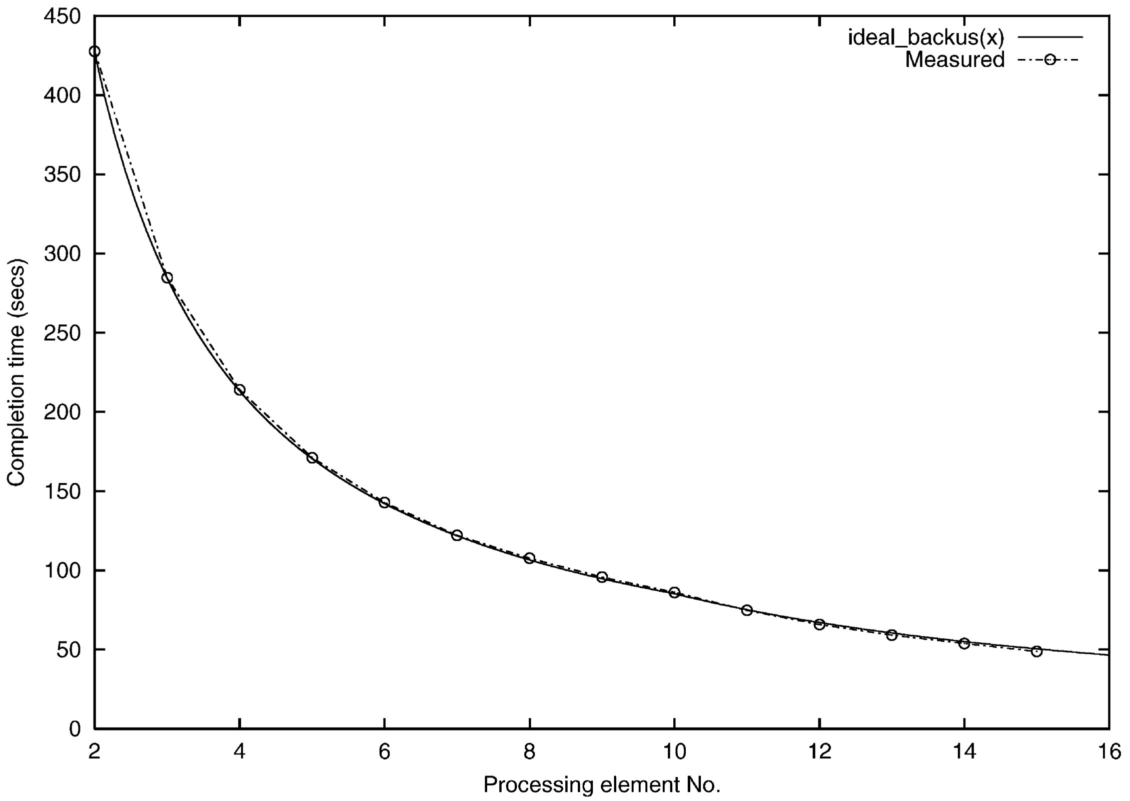


Fig. 5. Ideal vs. measured completion time.

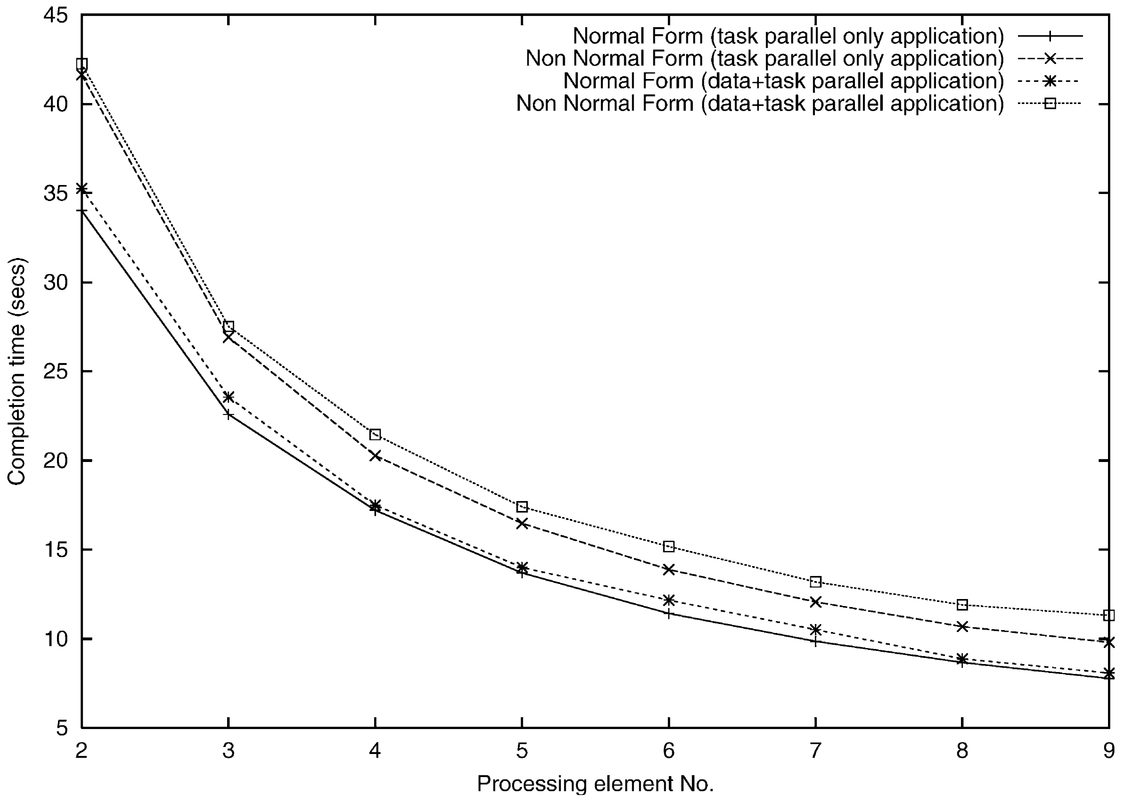


Fig. 6. Normal vs. non-normal form completion times, for two different "synthetic" applications.

In the local task pool manager a thread is forked for each one of the remote hosts used to compute the skeleton program. Such thread obtains a local reference to a remote RMI server, first; then issues a `setRemoteWorker` remote method call in order 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,[8] asks the remote server to compute the MDFi by issuing a remote `execute` method call and deposits the result in the task pool (see Fig. 4).

The MDF graph obtained from the `JSkeletons` 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` class). Such optimization rules implement the "normal form" concept outlined in Section 3.

As the skeleton program is provided by the programmer as a single (possibly nested) `JSkeletons` 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 skeletons). Furthermore, the `Object[ ] getSkeletonInfo private` method of the `JSkeletons` 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 `JSkeletons` 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

[8] Using proper `TaskPool synchronized` methods.

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 [14].

6. Experiments

In order to assess Lithium performance, we performed a set of experiments on a Beowulf class Linux cluster operated at our department, as well as on a set of "production" workstations available at our department.

The cluster based experiments were aimed at demonstrating Lithium performance features, mainly. The cluster used for the experiments hosts 17 nodes: one node (`backus.di.unipi.it`) is devoted to cluster management, code development and user interface. The other 16 nodes (ten 266 MHz Pentium II and six 400 MHz Celeron nodes) are exclusively devoted to parallel program execution. All the nodes are interconnected by a (private, dedicated) switched Fast Ethernet network. `backus` is a dual hosted node and provides access to the rest of the cluster node from Internet hosts. All the experiments have been performed using Blackdown JDK ports version 1.2.2 and 1.3.

First of all we considered the overhead introduced by serialization. As data flow tokens are dispatched to remote executor processes exploiting plain Java serialization, and as we use Java `Vector` objects to hold tokens, we measured the size overhead of the `Vector` class. The experiments showed that serialization does not add significant amounts of data to the real user data ($<10\%$) and therefore serialization does not cause significant additional communication overhead.

Second, we measured the Lithium applications absolute completion time. Typical results are drawn in

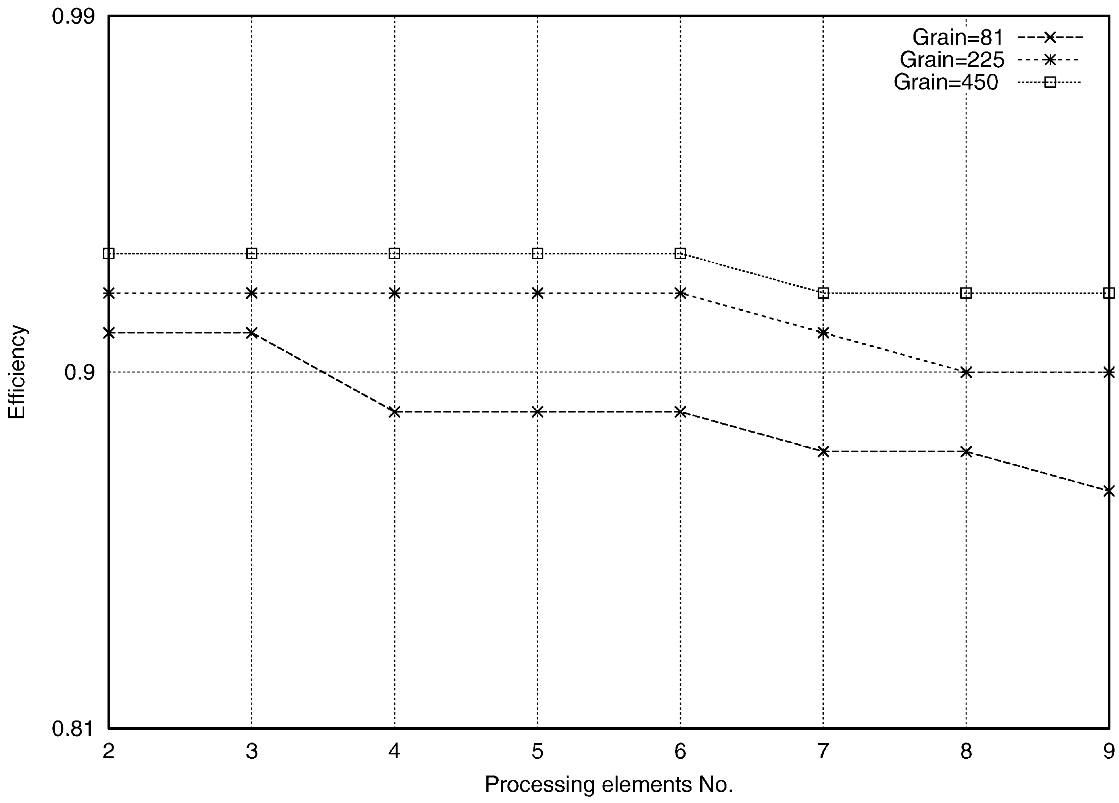


Fig. 7. Effect of grain on efficiency.

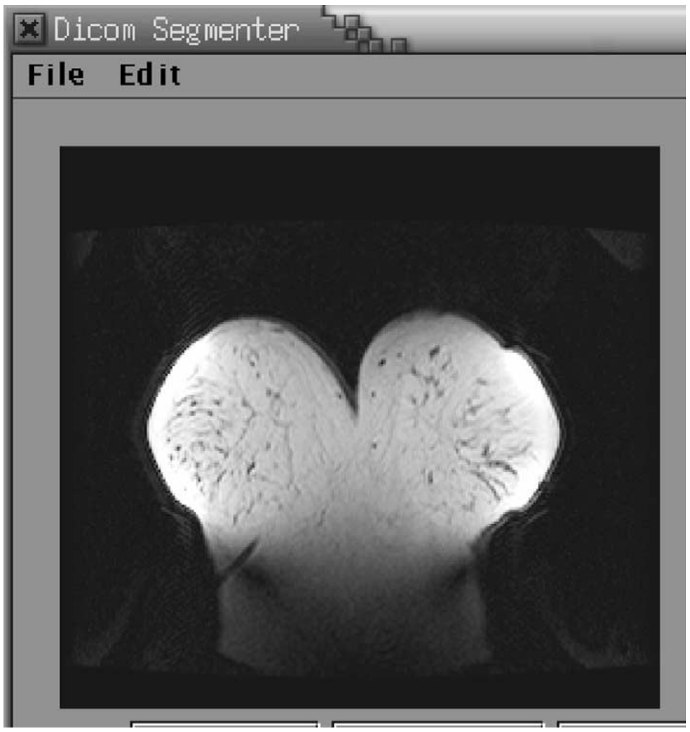


Fig. 8. Medical image segmentation application: screen snapshot.

Fig. 5. The plot shows that Lithium support scales. The graph actually refers to a skeleton version of a Mandelbrot benchmark, but we achieved similar results also with simple numerical applications. The completion times (ideal and measured) show an additional decrement from 10 nodes onwards, as the 11–16th nodes are more powerful that the first 10 nodes and therefore take a shorter time to execute sequential portions of Java code.

Third, we measured the impact of normal form optimizations, exploiting the possibility provided by Lithium of asking the execution of either the original program or the (automatically derived) normal form one. Fig. 6 plots the differences in the completion time of different applications executed using normal and non-normal form. As expected normal form always performs better that non-normal form. The good news are that it performs significantly better and *scales* better (non-normal form program stops scaling before normal form ones).

In addition, we measured the effect of computational grain of MDFi on efficiency (Fig. 7).[9] Computational 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 (via plain Java RMI). Experimental results showed that efficiency is always more than 90% when grain is higher than 100 (roughly). When average grain is under 100 efficiency falls under 90% already when three processing elements are used for program execution and continues to decrease rapidly as more and more processing elements are used.

We also performed experiments using a real application. We considered a medical application rendering

[9] As usual, we define efficiency (ϵ) as $\epsilon = T_{\text{seq}}/(nT_{\text{par}}(n))$, where T_{seq} represents the sequential execution time, n is the parallelism degree and $T_{\text{par}}(n)$ the time spent in the execution of the parallel program with parallelism degree n.

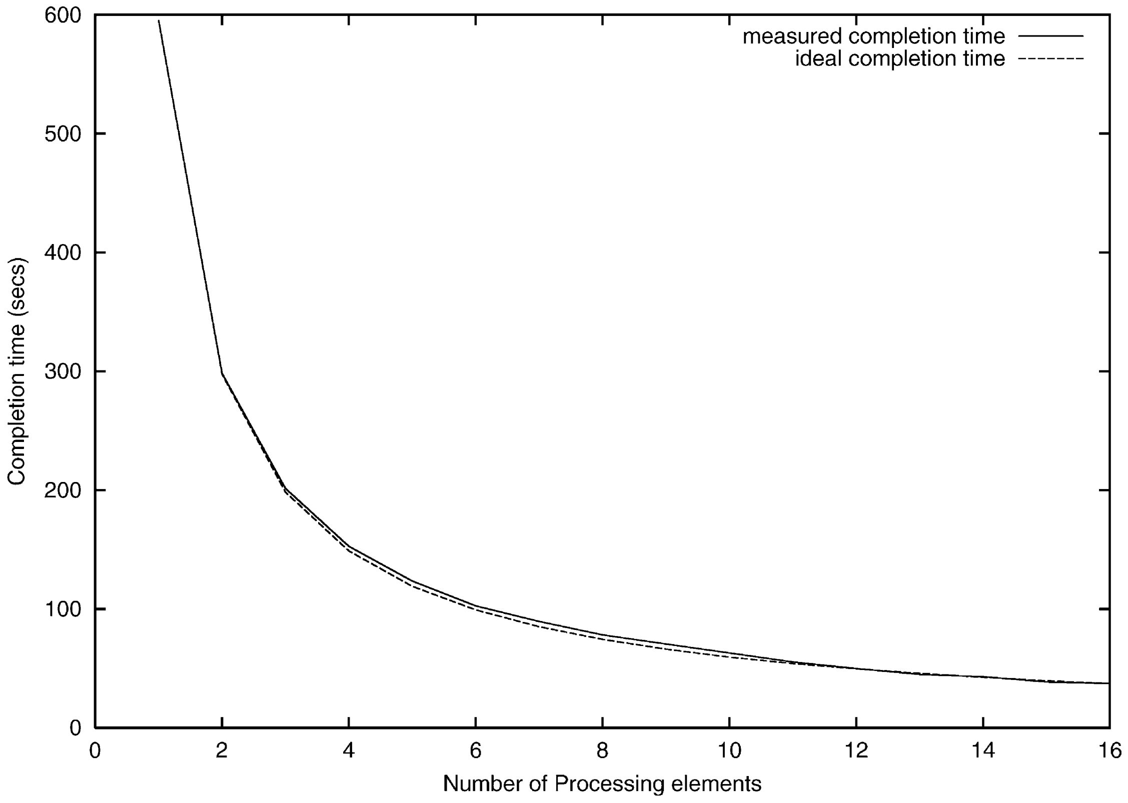


Fig. 9. Results with the medical image segmentation application.

mammography images. Images come from patient analysis. First, a set of images is taken, each representing a breast slice, more and more distant from patient ribs. Then some series of similar images are taken after the ignition of a contrast liquid. Overall a single, complete examine consists of about 100 images. Each image must be properly segmented in order to evidentiate the interest zones (i.e. zones where cancer may be discovered). Fig. 8 shows one of such images in a video snapshot taken from our Java segmentation application. The rendering of an image set takes about 10 min on a 266 MHz Pentium II Linux box. Fig. 9 shows the times achieved using Lithium on our cluster. The application perfectly scales[10] and efficiency is constantly over 90% in this case. Fig. 10 plots efficiency of segmentation application. Superscalar efficiency in the right part of the plot is due to the fact that sequential times are taken onto the Pentium processors (processors from 1 to 10) and processors 11–16 happen to be faster.

With the medical image (Fig. 11) segmentation code we also performed experiments on our department production workstations, in order to assess the load balancing policies of the MDF execution mechanism. The production workstations used range from 233 MHz Pentium II Linux boxes to dual 450 MHz Pentium III and 1.6 GHz Pentium IV Linux workstations. All the machines are interconnected by means of a plain 10 Mb (not switched) Ethernet network that happens to be very busy all the time.

The first result is that using faster machines Lithium achieves better completion times. The processing of 100 images took about 1.71 min on our 266 MHz Pentium II cluster on six PEs, while using six faster production workstations (three 1.2 GHz Pentium IV, two

[10] The data is relative to normal form execution. The application is a farm with three stage workers.

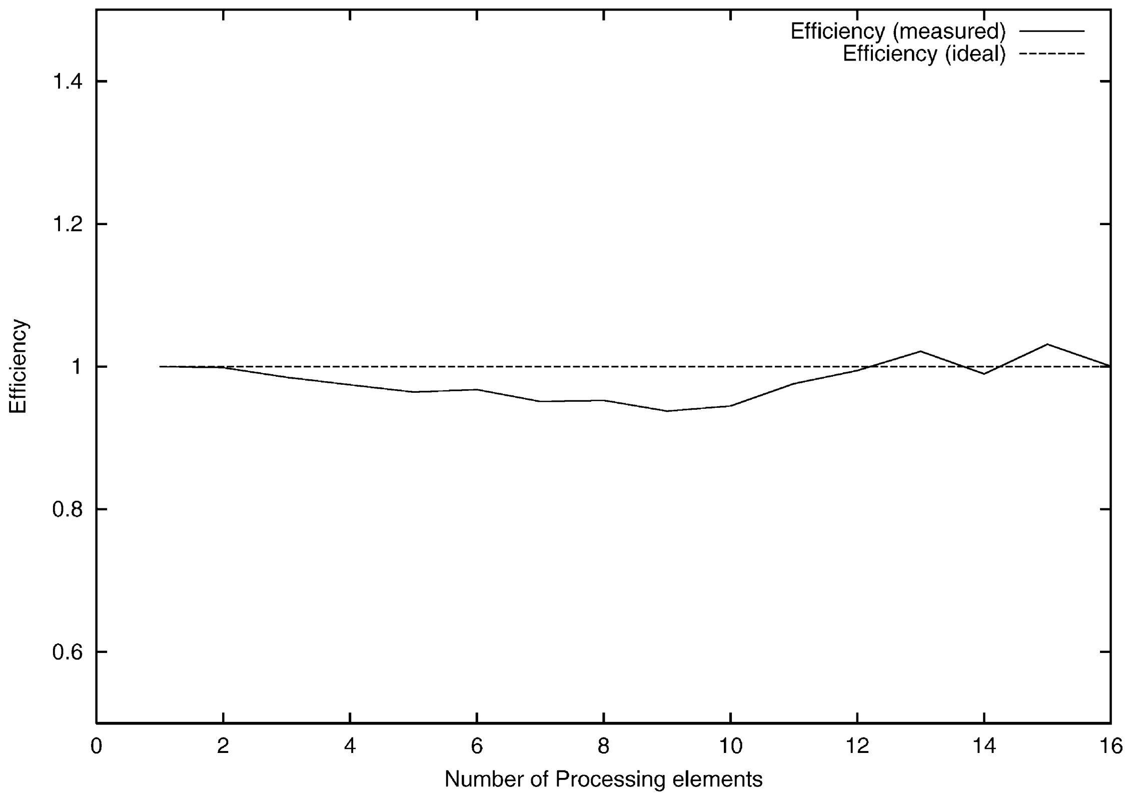


Fig. 10. Efficiency of medical image segmentation application. The efficiency is figured out with respect to the sequential execution time computed on the slower processors (PE $\in [1, 10]$).

450 MHz Pentium III and a single 233 MHz Pentium II machine) the same processing took only 0.85 min.

The second result concerns load balancing. Fig. 12 shows the number of task executed by each workstation along with workstation bogomips (the rough performance measure taken by Linux kernel at boot time) and load (measured with `uptime` during the experiments). It is clear that load balancing has been achieved in that slower, more loaded workstation participated in the computation by computing a smaller number of tasks with respect to faster, unloaded workstations.

7. 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 CO_2P_3S one [21,26]. Actually this project derives from the design pattern experience [27]. 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 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

```
import lithium.*;

public class Emitter extends JFrame {
    ...
    public static void main(String [] args) {
        ...
        File dir = new File(args[0]);
        String[] files = dir.list();
        ...
        JSkeletons stage1 = new DicomToImage();
        JSkeletons stage2 = new Segmenter();
        JSkeletons stage3 = new PostProcessing();
        JSkeletons worker = new Pipe();
        worker.addWorker(stage1);
        worker.addWorker(stage2);
        worker.addWorker(stage3);
        Farm theMain = new Farm(worker);
        Ske eval = new Ske();
        String[] hosts = {"izar","icaro","quanto",
                          "cassiopea","montecristo"};
        eval.addHosts(hosts);
        eval.setOptimizations();
        eval.skelProgram(theMain);
        for(int f=0; f<taskNo; f++) {
            ...
            b = f(DD.getImageToDisplay(files[f]));
            try {
                eval.setupTaskPool((Object)(b));
            } catch (Exception e) {
                System.out.println("TP setup: "+e);
                return;
            }
        }
        eval.stopStream();
        try {
            eval.parDo();
        } catch (Exception e) {
            System.out.println("Start: "+e);
            return;
        }
        while(!ske.isResEmpty()) {
          try {
             res = (byte[]) eval.readTaskPool();
          } catch (Exception e) {
             System.out.println("Result fetch: "+e);
             return;
          }
          ...
          e.doImage(g(res),w,h);  // display image
        }
        return;
    }
```

Fig. 11. Medical image processing application skeleton.

to our one, especially in that it does not use any kind of MDF 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

		WS_1	WS_2	WS_3	WS_4	WS_5	WS_6
Bogomips		3204	3630	901	3204	466	897
6 PE	load	1.44	0.75	1.90	0.24	0.39	0.34
	tasks#	20	29	5	26	7	13
4 PE	load	1.44	0.88	0.46	0.37	–	–
	tasks#	24	36	6	34	–	–

Fig. 12. Load balancing on production, heterogeneous processing elements. First row reports workstation bogomips. Rows 2, 3 and 4, 5 are relative to a six PE and a four PE run (100 tasks), respectively.

parallel programming environment that can be used to derive high performance parallel Java code running on parallel/distributed machines.

MDF implementation techniques have also been used to implement skeleton based parallel programming environments by Serot in the Skipper project [12,28]. 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 environment (the whole Skipper environment is written using Ocaml, the ML implementation from INRIA).

8. Conclusions and future work

We described a new Java parallel programming environment providing the programmer with simple tools suitable to develop efficient parallel programs on workstation networks/clusters. Lithium is the first skeleton based parallel programming environment written in Java and implementing skeleton parallel execution by using MDF techniques. We performed experiments that demonstrate that good scalability and efficiency figures can be achieved. Lithium has been released as open source and it is currently available as open source at http://massivejava.sourceforge.net. Our group is currently investigating the possibility to add two new features to Lithium: on the one hand, we plan to introduce in Lithium all the security features needed to use it on wide area networks (e.g. the Internet). Workstation clusters represent a sort of "protected" environment, therefore we have implemented no particular security policy (e.g. to check the accesses performed on the RMI servers acting as MDFi executors in Lithium). On the other hand, we plan to replace the current centralized MDFi task pool repository of Lithium by a *distributed* repository, to avoid bottlenecks in the fireable instruction accesses.

Acknowledgements

We wish to thank Emilio Tuosto and Fabio Gadducci for the useful discussions concerning Lithium operational semantics. This work has been partially supported by the ASI-PQE2000 Programme Development of Applications for Earth Observation with High-Performance Computing Systems and Tools.

References

[1] Sun, The Java home page, 2002. http://java.sun.com.
[2] JavaGrande, The JavaGrande home page, 2002. http://www.javagrande.org.
[3] D.C. Hyde, Java and different flavors of parallel programming models, in: R. Buyya (Ed.), High Performance Cluster Computing, Prentice-Hall, Englewood Cliffs, NJ, 1999, pp. 274–290.
[4] L.M. Silva, Web-based parallel computing with Java, in: R. Buyya (Ed.), High Performance Cluster Computing, Prentice-Hall, Englewood Cliffs, NJ, 1999, pp. 310–326.
[5] G. Antoniu, L. Bougé, P. Hatcher, M. MacBeth, K. McGuigan, R. Namyst, Compiling multithreaded Java bytecode for distributed execution, in: A. Bode, T. Ludwig, W. Karl, R. Wismuller (Eds.), EuroPar'2000 Parallel Processing, LNCS, No. 1900, Springer, Berlin, 2000, pp. 1039–1052.
[6] Y. Aridor, M. Factor, A. Teperman, cJVM: a single system image of a JVM on a cluster, in: Proceedings of the International Conference on Parallel Processing, Fukushima, Japan, 1999. http://www.haifa.il.ibm.com/projects/systems/cjvm/index.html.
[7] MpiJava, The MpiJava home page, 2001. http://www.npac.syr.edu/projects/pcrc/mpiJava/.
[8] jPVM, The jPVM home page, 2001. http://www.chmsr.gatech.edu/jPVM/.
[9] M. Cole, Algorithmic Skeletons: Structured Management of Parallel Computations, Research Monographs in Parallel and Distributed Computing, Pitman, London, 1989.
[10] P. Au, J. Darlington, M. Ghanem, Y. Guo, H. To, J. Yang, Co-ordinating heterogeneous parallel computation, in: L. Bouge, P. Fraigniaud, A. Mignotte, Y. Robert (Eds.), Europar'96, Springer, Berlin, 1996, pp. 601–614.
[11] B. Bacci, M. Danelutto, S. Pelagatti, M. Vanneschi, SkIE: a heterogeneous environment for HPC applications, Parallel Comput. 25 (1999) 1827–1852.
[12] J. Serot, D. Ginhac, R. Chapuis, J. Derutin, Fast prototyping of parallel-vision applications using functional skeletons, Mach. Vision Appl. 12 (2001) 217–290, Springer, Berlin.
[13] M. Danelutto, R.D. Cosmo, X. Leroy, S. Pelagatti, Parallel functional programming with skeletons: the OCAMLP3L experiment, in: Proceedings of the ACM Sigplan Workshop on ML, 1998, pp. 31–39.
[14] M. Danelutto, M. Stigliani, SKElib: parallel programming with skeletons in C, in: A. Bode, T. Ludwing, W. Karl, R. Wismüller (Eds.), EuroPar'2000 Parallel Processing, LNCS, No. 1900, Springer, Berlin, 2000, pp. 1175–1184.
[15] H. Kuchen, A skeleton library, Technical Report 6/02-I, Angewandte Mathematik und Informatik, University of Munster, 2002.
[16] S. Pelagatti, Structured Development of Parallel Programs, Taylor & Francis, London, 1998.
[17] M. Danelutto, Task farm computations in Java, in: Buback, Afsarmanesh, Williams, Hertzberger (Eds.), High Performance Computing and Networking, LNCS, No. 1823, Springer, Berlin, 2000, pp. 385–394.
[18] M. Aldinucci, M. Danelutto, Stream parallel skeleton optimisations, in: Proceedings of the IASTED International Conference Parallel and Distributed Computing and Systems, IASTED/ACTA Press, Boston, 1999, pp. 955–962.
[19] M. Aldinucci, Automatic program transformation: the meta tool for skeleton-based languages, in: S. Gorlatch, C. Lengauer (Eds.), Constructive Methods for Parallel Programming, Advances in Computation: Theory and Practice, NOVA Science Publisher, New York, 2002, pp. 59–78, draft available at ftp://ftp.di.unipi.it/pub/Papers/aldinuc/meta_book_dft.ps.gz).
[20] M. Aldinucci, M. Danelutto, An operational semantics for skeletons, Technical Report TR-02-13, Department of Computer Science, University of Pisa, Italy, July 2002. http://www.di.unipi.it/ricerca/TR/.
[21] S. McDonald, D. Szafron, J. Schaeffer, S. Bromling, Generating parallel program frameworks from parallel design patterns, in: A. Bode, T. Ludwing, W. Karl, R. Wismüller (Eds.), EuroPar'2000 Parallel Processing, LNCS, No. 1900, Springer, Berlin, 2000, pp. 95–105.
[22] P. Teti, Lithium: a Java skeleton environment, Master's Thesis, Department of Computer Science, University of Pisa, October 2001 (in Italian).
[23] M. Danelutto, Dynamic run time support for skeletons, in: E.H. D'Hollander, G.R. Joubert, F.J. Peters, H.J. Sips (Eds.), Proceedings of the International Conference ParCo99, Vol. Parallel Computing Fundamentals and Applications, Imperial College Press, 1999, pp. 460–467.
[24] M. Danelutto, Efficient support for skeletons on workstation clusters, Parallel Process. Lett. 11 (1) (2001) 41–56.

[25] C. Nester, R. Philippsen, B. Haumacher, A more efficient RMI for Java, in: Proceedings of the ACM 1999 JavaGrande Conference, 1999, pp. 152–157.

[26] S. MacDonald, J. Anvik, S. Bromling, J. Shaeffer, D. Szafron, K. Tan, From patterns to frameworks to parallel programs, Parallel Comput. 28 (11) (2002) 1685–1708.

[27] E. Gamma, R. Helm, R. Johnson, J. Vissides, Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley, Reading, MA, 1994.

[28] J. Serot, D. Ginhac, Skeletons for parallel image processing: an overview of the SKiPPER project, Parallel Comput. 28 (11) (2002) 1663–1683.

M. Aldinucci graduated at the University of Pisa in 1997 in Computer Science and is currently PhD student at the Department of Computer Science, University of Pisa. His main research interests are in parallel/distributed computing. In particular, in the field of structured parallel programming models and distributed shared memory systems.

M. Danelutto received his PhD in 1990 from University of Pisa and is currently an Associate Professor at the Department of Computer Science, University of Pisa. His main research interests are in parallel/distributed computing. In particular, in the field of structured parallel programming models and coordination languages. He was one of the designers of the skeleton based parallel programming language P3L. He is currently involved in several national research projects aimed at designing structured coordination languages for high performance parallel/distributed/GRID computing.

P. Teti was born in 1977. He graduated at the University of Pisa in 2001 in Computer Science. He is currently a consultant in the field of embedded hard real time Linux systems.