JaSkel: A Java Skeleton-Based Framework for Structured Cluster and Grid Computing*

J. F. Ferreira, J. L. Sobral, and A. J. Proença
Departamento de Informática, Universidade do Minho, Braga, Portugal

Abstract

This paper presents JaSkel, a skeleton-based framework to develop parallel and grid applications. The framework provides a set of Java abstract classes as a skeleton catalogue, which implements recurring parallel interaction paradigms. This approach aims to improve code efficiency and portability. It also helps to structure scalable applications through the refinement and composition of skeletons. Evaluation results show that using the provided skeletons do contribute to improve both application development time and execution performance.

1. Introduction

Cluster and Grid computing environments require adequate tools to structure scalable applications, taking advantage of the underlying multi-layer architecture, namely a grid of clusters with shared memory multi-core processing nodes.

Skeleton based tools are especially attractive for these environments, since they provide a structured way to develop scalable applications, supporting composition of base skeletons. This work addresses the use of skeletons applied to a Java object-oriented environment.

Skeletons are abstractions modelling common, reusable parallelism exploitation patterns [2][3]. A skeleton may also be seen as a high order construct (i.e. parameterized by other pieces of code) which implements a particular parallel behaviour.

Our skeleton catalogue is a collection of code templates implemented as a library of Java abstract classes. The catalogue aims to help programmers to create code for grids of parallel computing platforms.

The computing development environment presented here contains a Java skeleton-based framework, JaSkel, as a support structure where software projects can be organised and developed. Programmers select appropriate skeletons and fill in the gaps with pieces of domain-specific code. The environment provides all the relevant components for scalable computing platforms, including a run-time adaptive load distribution.

This work differs from other research environments [9][10] in the way it uses different and orthogonal components for distinct tasks: a skeleton-based framework to structure scalable applications (which may use one or more processors), a code generator which supports distribution of selected object classes and an adaptive run-time load and data scheduler. The independence between these components lets programmers develop, test and run structured applications in a non-distributed environment, and it simultaneously supports an efficient use of skeletons on distributed and shared memory architectures.

JaSkel also exhibits advantages over competitive skeleton frameworks: JaSkel is an inheritance based framework and it supports orthogonal and hierarchical composition of skeletons.

Domain-specific code in skeletons is specified by refinement, implementing abstract methods. This structure overcomes the lack of extensibility of other skeleton frameworks, since the provided skeletons can be modified by inheritance. The framework itself is organised as a hierarchy of classes, e.g., a concurrent farm extends a sequential farm.

Complex parallel applications can be obtained through composition of skeletons, such as multi-level farms or farms of pipelines. This feature may take advantage of current grid computing, where grid nodes are clusters with SMP nodes.

The description of the code generator and run-time load distribution tools is out of the scope of this presentation.

The remainder of this paper is organised as follows. Section 2 presents related work. Section 3 describes the skeleton-based Java framework built to help programmers to structure scalable applications. Section 4 presents qualitative and performance evaluation of skeletons from the framework. The last section discusses the obtained results and draws some conclusions on the work done so far.

*This work was supported in part by PPC-VM project (POSI/CHS/47158/2002) and by project SeARCH (contract REEQ/443/2001) both funded by Portuguese FCT.

2. Related work

Several skeleton based systems have been proposed and most support skeleton composition (i.e., nesting) [11][5][3]. The most relevant Java environments for parallel programming based on skeletons are Lithium [10] and CO_2P_3S [9]. Lithium supports skeletons for pipeline, farm and divide & conquer, as well as other low level skeletons (map, composition). CO_2P_3S is based on generative patterns, where skeletons are generated and the programmer must fill the provided *hooks* with domain specific functionality.

The main differences between JaSkel and these approaches are:

- JaSkel explores class hierarchy and inheritance along with object composition;
- JaSkel provides separate tools to structure parallel applications and to implement communication and distribution;
- JaSkel includes sequential and parallel skeletons.

A skeleton hierarchy overcomes the main limitations of other alternatives, supporting refinement of provided skeletons, including implementation of non-pure functional skeletons (e.g., skeletons that maintain state between invocations).

A separate tool to implement distribution provides an enhanced fine-grained control over distribution issues. Distribution is orthogonal to the framework class hierarchy, which lets the programmer to use several distribution middleware along with class hierarchies. This feature is required to address current grid heterogeneity, both in hardware and in middleware.

JaSkel also provides sequential versions of all skeletons; moving from a sequential version to a parallel one just requires the change of the name of the extended base class. This approach eases the development in early coding phases and in debugging activities.

3. A Java skeleton-based framework

Many parallel algorithms share the same generic patterns of computation and interaction. Skeletal programming proposes that such patterns be abstracted and provided as a programmer's toolkit. We call these abstractions *algorithmical skeletons*, *parallel skeletons* or simply *skeletons*.

Skeletons provide simple interfaces to programmers with an incomplete structure that can be parameterized by the number of processors, domain-specific code or data distribution; programmers can focus on the computational side of their algorithms rather than the control of the parallelism.

3.1. JaSkel framework

The current JaSkel prototype (JaSkel 1.01) includes skeletons for farm and pipeline parallel coding. To write a parallel application in JaSkel, a programmer must perform the following steps:

1) To structure the parallel program and to express it using the available skeletons;
2) To refine the supplied abstract classes and write the domain-specific code;
3) To write the code that starts the skeleton, defining other relevant parameters (number of processors, load distribution policy, ...).

The current JaSkel prototype provides several versions of farm and pipeline skeletons:

- a sequential farm;
- a concurrent farm that creates a new thread for each worker;
- a dynamic farm, which only sends data to workers when they require them;
- a sequential pipeline;
- a concurrent pipeline, which creates a new thread for each data flow.

A JaSkel skeleton is a simple Java class that implements the *Skeleton* interface and extends the *Compute* class (Figure 1). The *Skeleton* interface defines a method *eval* that must be defined by all the skeletons. This method starts the skeleton activity.

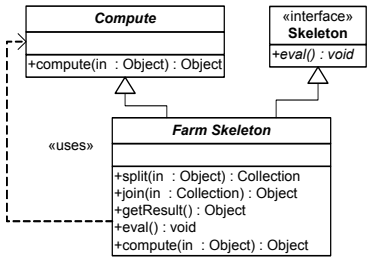

Figure 1. The sequential farm skeleton implements the Skeleton interface and extends the Compute class to allow skeleton composition.

To create objects that will perform domain-specific computations, the programmer must create a subclass of class *Compute* (inspired in Lithium). The *Compute* abstract class defines an abstract method *public abstract Object compute(Object input)* that defines the domain-specific computations involved in a skeleton.

For instance, to create a Farm (see Figure 1 for a Farm UML class diagram), a programmer needs to perform the following steps:

1) To create the worker's class, extending *Compute* and implementing the inherited method *public Object compute(Object input)*;
2) To create the master's class, extending *Farm*, defining the methods *public Collection*

split(Object initialTask) and *public Object join(Collection partialResults)*;

3) To create a new instance of the master's class and call the method *eval*; this method will basically perform the following steps:
   - it creates multiple workers;
   - it splits the initial data using the defined *split* method;
   - it calls *compute* method from each worker with the pieces of data returned by method *split*;
   - it merges the partial results using the defined *join* method.

The specialisation or the creation of a new skeleton is done by class refinement. The concurrent farm skeleton extends the sequential farm skeleton and a dynamic farm extends a concurrent farm.

JaSkel skeletons are also subclasses of *Compute* class to allow composition. The method *public Object compute(Object input)* on skeletons calls the *eval* method to start the skeleton activity.

3.2. Building JaSkel applications

The best way to show how to build a skeleton-based application is through an example: to find and count all prime numbers up to *N*.

A Java implementation that codes this algorithm marks the multiples, setting them to 0. This implementation consists of two entities: a number generator and a prime filter. The first generates the input integer array [2..N] and the latter filters the non-prime integers. In a simple farm parallelisation the input array is decomposed in smaller pieces, and each piece is sent to a prime filter; each prime filter will test the input integers using the filter [2..sqrt(N)].

The examples bellow show how the JaSkel framework codes this algorithm using different scalable structures. The code was slightly simplified to improve readability.

3.2.1. Prime sieve as a farm. The prime filter (the farm worker) is illustrated in Code 1. Its main method is *filter* (line 03), which filters the given integer array. The *compute* method (lines 06-08), needed to define the skeleton's domain-specific code, delegates its job to the method *filter*. Note that the class *FarmPrimeFilter* is a subclass of *Compute* (line 01).

```
01  public class FarmPrimeFilter extends Compute {
02      …
03      public int[] filter(final int[] num) {
04          … // removes non-primes from the list
05      }
06      public Object compute(Object input) {
07          return this.filter((int[]) input);
08  }
```

Code 1.The FarmPrimeFilter class.

Code 2 illustrates the class *GeneratorFarm* (the farm master): it extends the skeleton *FarmConcurrent* (line 1), it uses private methods to implement methods *split* (lines 5-7) and *join* (lines 8-10).

```
01  public class GeneratorFarm extends FarmConcurrent {
02      public GeneratorFarm(Computer worker, Object inputTask) {
03          super(worker, inputTask);
04      }
05      public Collection split(Object initialTask) {
06          return(Packs.split((int[])initialTask,blocksize));
07      }
08      public Object join(Collection partialResults) {
09          return(Packs.join((Vector) partialResults));
10      }
11  }
```

Code 2. The generator farm class.

Code 3 shows the code that connects these entities:

- it creates one prime filter object (line 3);
- it creates a new farm generator object, setting its parameters: the worker, and input data (line 4);
- it starts the skeleton activity, calling method *eval*;
- it gets the final result, using method *getResult*.

```
01  int [] ar = new int[…];      // buffer of numbers to filter
02  for(int i=min; i<=max; i+=2)  ar[(i-min)/2]=i; // list of numbers to filter
03  FarmPrimeFilter pf = new FarmPrimeFilter();       // create one filter
04  GeneratorFarm g = new GeneratorFarm(pf, ar);        // farmer
05  g.eval();                          // starts the farming process
06  Object o = g.getResult();        // get results
```

Code 3. The main farm code.

3.2.2. Prime sieve as a farm of farms. A two-level farming can be easily created with farm nesting: only a few modifications are required in lines 3-4 to create this hierarchy (Code 4). The line 4 in Code 4 creates one inner farm in the same way as the previous example. Line 5 creates the main farm, where each worker is also a farm. The JaSkel framework will clone the inner farm and its workers on each node.

```
…  // same as code 3
03   FarmPrimeFilter pf = new FarmPrimeFilter();
04   GeneratorFarm innerFarm = new GeneratorFarm(pf, null);
05   GeneratorFarm g = new GeneratorFarm(innerFarm, ar);
… // same as code 3
```

Code 4. A two level farm.

4. Performance evaluation

This section aims to show that the benefits of the skeleton based framework did not impose performance degradation. We present the performance of a reference algorithm: a parallel ray tracer, from the Java Grande Forum [6]. These results were collected on a cluster with 16 dual Xeon 3.2 GHz 2MB L2 cache computing nodes, with multi-threading enabled (i.e., with four Intel HT virtual processors), connected by Gigabit Ethernet, running CentOS 4.0. Presented values are the median value of 5 runs.

Table 1 compares executions times and speed-ups of two implementations (with an image of size 500x500):

a version converted to the MPP package (www.math.uib.no/~bjornoh/mtj/mpp), which performs close to the original mpiJava version [1] and a JaSkel version that also uses MPP. Both implementations place one worker on a physical CPU. Speed-ups are relative to the JGF sequential version. The object distribution in JaSkel follows the technique presented in [8]. The MPP package is fully written in Java (using java.nio) which avoids the instability problems in Java bindings to MPI, with performance values close to MPI implementations.

Table 1. Farm executions times (s): MPP and JaSkel version.

Workers (CPU)	4	8	16	24	32
MPP	17.18	8.71	4.53	3.04	2.33
JaSkel	17.28	8.76	4.45	2.96	2.28
MPP speed-up	4.0	7.8	15.0	22.4	29.3
JaSkel speed-up	3.9	7.8	15.3	23.0	29.9

The second test (Table 2) compares the execution times and speed-ups of a single level JaSkel farm against a two level farm when running the RayTracer with a lower computation/communication ratio (obtained with a 75x75 pixels image). In the two level farm, each second level farm master has two workers, all placed on the same computing node.

Table 2. Farm executions times (s): JaSkel single level and two level farm

Workers (CPU)	8	16	24	32
Single level farm	0.249	0.160	0.138	0.138
Two-level farm	0.246	0.157	0.133	0.128
Single level speed-up	6.27	9.76	11.31	11.31
Two-level speed-up	6.35	9.94	11.74	12.20

The single level JaSkel farm suffers from excessive communication on a high number of nodes, which reduces the performance gain when the number of CPU increases. Single level farms in mpiJava and MPP versions present identical behaviour (results not shown).

A two level farm has fewer inter-node messages, leading to lower communication costs, which make this version attractive in computing clusters with multi-core and multi-CPU nodes.

These results also suggest that multi-level farms may more efficiently take advantage of multi-layered architectures, namely, grids of clusters, where computing nodes in cluster are moving towards platforms with multi-core CPUs. Hierarchically composed skeletons can be an effective way to deal with different interconnection latencies and bandwidths, which is one of the main difficulties when developing parallel application targeted for a grid of clusters environment.

5. Conclusion

Jaskel framework is a component of a computing development environment, based on a catalogue of skeletons. These are provided as abstract classes which can be further refined, or hierarchically grouped, to guide the development of structured parallel and grid applications of medium/large complexity. The skeleton framework aims to improve programmer's productivity and, at the same time, keeping high level of execution performance. Experimental time measurements on a ray tracer from JGF confirmed these expectations.

JaSkel framework has proved its usefulness and capabilities. Current work goes on developing more high level skeletons and methodologies to automatically tune some platform dependent parameters based on run-time information [7]. Full integration of JaSkel with the remaining components of the computing environment – the code generator and the dynamic task/data distribution – still require further tests to evaluate integrated grid environments.

6. References

[1] B. Carpenter, V. Getov, G. Judd, T. Skjellum, G. Fox, MPJ: MPI-like Message Passing for Java. *Concurrency: Practice and Experience*, vol. 12, n. 11. September 2000.

[2] D. Cole, *Algorithmic Skeletons: structured management of parallel computation*, Pitman/MIT press, 1989.

[3] F. Rabhi, S. Gorlatch, S. (ed), *Patterns and Skeletons for Parallel and Distributed Computing*, Springer, 2003

[4] G. Andrews, *Foundations of Multithreaded, Parallel, and Distributed Programming*, Addison Wesley, 2000.

[5] J. Darlington, Y. Guo, H. To, J. Yang. "Parallel Skeletons for Structured Composition", *ACM PPoPP'95*, Santa Clara, USA, 1995.

[6] J. Smith, J. Bull, J. Obdrzálek, "A Parallel Java Grande Benchmark Suite", *SC 2001*, Denver, November 2001.

[7] J. Sobral, A. Proença, "A SCOOPP Evaluation on Packing Parallel Objects in Run-time", *VecPar-2000*, Porto, July 2000, LNCS vol. 1981, Springer 2000.

[8] J. Sobral. "Incrementally Developing Parallel Applications with AspectJ", *IEEE IPDPS'06*, Rhodes, Greece, April 2006.

[9] K. Tan, D. Szafron, J. Schaeffer, J. Anvik, S. MacDonald, "Using Generative Design Patterns to Generate Parallel Code for a Distributed Memory Environment", *ACM PPoPP'03*, San Diego, California, USA, June, 2003.

[10] M. Aldinucci, M. Danelutto, P. Teti, "An advanced environment supporting structured parallel programming in Java", *Future Generation Computing Systems*, vol. 19, 2003.

[11] P. Trinder, K. Hammond, H. Loidl, S. Jones. "Algorithm + Strategy = Parallelism", *Journal of Functional Programming*, 8(1), January 1998.