Enabling JaSkel Skeletons for Clusters and Computational Grids

J. L. Sobral, A. J. Proença

Centro de Ciências e Tecnologias da Computação, Departamento de Informática
Universidade do Minho, Braga, Portugal
jls@di.uminho.pt

***Abstract*— JaSkel is a skeleton-based framework to develop efficient concurrent, parallel and Grid applications. It provides a set of Java abstract classes that implement recurring parallel interaction paradigms. The key feature of JaSkel is to use aspect-oriented external tools to address distributed execution, by injecting code to support communication middleware into JaSkel built-in skeleton implementations. This feature, when combined with the ability to develop nested skeletons, can help to tailor JaSkel applications to efficiently run on a Grid of clusters systems, by taking advantage of inter/intra-cluster and/or intra-node communications. This paper describes the JaSkel distributed execution tools and how they interplay with the JaSkel framework to transparently run applications on a wide range of computing platforms, from multi-core to computational Grids. Results are presented to show the feasibility and scalability of this approach.**

I. INTRODUCTION

The current move from higher clock frequency CPU devices into multi-core chips, together with the increased availability of federations of cluster systems, stressed the demand for concurrent and distributed applications. Current programming environments do not yet conveniently cope with this wide range of computing platforms. These environments are either targeted for shared memory systems, by providing thread-based approaches, or targeted for distributed environments, using message passing or remote method invocation. Adapting an application to Grid systems also requires a significant upfront investment, which is an obstacle to a faster embracing of Grids systems.

Developing applications for computational Grids requires applications that are aware of the hierarchical nature of a Grid to take advantage of multiple levels of locality: a Grid can be built of multiple clusters, where each cluster node can have several CPU/cores, and each level usually provides its type of middleware for inter-task communication. For instance, each Grid cluster usually provides a way for job submission, where all data must be provided through files, whereas intra-cluster communications can use more efficient inter-process communication middleware (e.g., MPI).

Skeletons are abstractions [1], [2] modelling common and reusable parallelism patterns. Their intended purpose is to hide low-level, platform dependent, details from the programmer. A skeleton based framework [3]-[5] usually provides a set of skeletons that are efficiently implemented on a range of computing platforms. The programmer can use the provided skeletons to build parallel applications without being aware of the skeletons implementation.

Skeleton based approaches are especially attractive for Grid environments as they support composition of skeletons [6] (i.e., skeleton nesting) that can closely match the hierarchical nature of a Grid. For instance, a two-level farm can take advantage of a Grid of clusters by deploying an inner farm on each Grid cluster, and using the top level farm to coordinate work allocation among Grid clusters.

JaSkel [5] is a skeleton framework that provides several skeletons and that supports skeleton nesting. It provides a collection of code templates implemented as a library of Java abstract classes. The collection aims to help programmers to create parallel applications that can run on a wide range of computing platforms.

JaSkel differs from other skeleton approaches [3], [4], [7], [8] in the way it deals with distribution concerns (i.e., to adapt skeletons to a specific communication middleware): these are solved in an orthogonal way, by code generators that support distribution of selected skeleton classes. The JaSkel framework presented in [5] already had built-in support for multi-threaded codes, while this communication focuses on complementary distribution tools and aims to show their overall advantage in scaling applications to a wide range of platforms and in multi-level computing systems. These tools generate and launch two types of distributed skeletons: one, where skeletons communicate through message passing or remote method invocations; another, where skeletons submit jobs to clusters/Grids through a resource manager system (such as PBS). These distribution tools, when combined with a multi-level farm, generate application code that takes advantage of the multi-level nature of a Grid of cluster systems.

The remainder of this paper is organised as follows. Section 2 presents related work. Section 3 provides an overview of the skeleton-based Java framework. Section 4 describes the two developed distribution tools. Section 5 presents performance evaluation results and the last section draws conclusions on the work done so far pointing future directions of research.

II. RELATED WORK

Several skeleton based approaches are described in the literature that also support skeleton composition [2], [6], [8]. Lithium [3] and CO2P3S [4] are two Java-based skeleton environments for parallel programming. Lithium supports skeletons for pipeline, farm and divide & conquer, as well as other low level skeletons (map, composition). CO2P3S is based on generative patterns, where skeletons are generated and the programmer must fill the provided hooks with domain specific functionality. HOCs [7] were specifically designed for Grid environments by supporting independent deployment of skeleton and application specific code.

The most relevant difference of JaSkel from these other approaches is the use of external tools to adapt skeletons to a specific distributed environment (i.e., communication middleware). This approach presents several advantages over competitive approaches:

- skeletons can scale to a wide range of computing platforms, from multi-core systems to computational Grids;
- JaSkel skeletons can combine multiple communication middleware into a single application taking advantage of the layered composition of clusters and Grids;
- distribution code, required to adapt skeletons to a specific distributed environment, does not impose performance constrains when it is not required, since it is not include in the build;
- JaSkel skeletons do not need to provide distribution specific hooks (e.g., skeleton factories).

Additionally, JaSkel explores class hierarchy and inheritance along with object composition and includes sequential and parallel skeletons. A skeleton hierarchy overcomes some limitations of other alternatives, supporting refinement of the provided skeletons. JaSkel includes sequential versions of all skeletons; moving from a sequential skeleton to a parallel implementation simply entails changing the extended base class. This approach eases the development in early coding phases and in debugging activities.

Most Java based approaches for parallel computing are concerned to Java distributed machines aimed to address distributed thread execution. Examples are cJVM [9], Hyperion [10], Jakal [11] and JESSICA2 [12]. Java Party [13], ProActive [14] and Ibis [15] are similar approaches but they rely on distributed objects. ProActive and Ibis also support the deployment of applications into Grid platforms, by including middleware that supports Grid environments. These approaches share with JaSkel the benefit of transparently supporting the execution of an application on shared and distributed memory systems. However these approaches do not present the benefits of skeleton approaches, namely the more structured application development, including the support for skeleton composition.

III. JaSkel Overview

Skeleton based frameworks provide a set of pre-defined structures (called skeletons) to develop parallel applications. The programmer selects the skeletons better suited for his/her concrete application, provides application specific code and the framework takes care of concurrency and distribution issues. By using skeletons the programmer can focus on the computational side of their algorithms rather than on parallelisation issues.

Current JaSkel framework provides several skeletons, including, pipeline, farm and heartbeat. Each of these skeletons implements a frequently occurring structure of parallel applications. To write an application in JaSkel the programmer must go through three steps:

1. Select the skeleton(s) that better fits the application parallelisation.
2. Refine the skeleton abstract class, providing the domain specific code, by filling skeleton hooks (i.e., abstract methods).
3. Write code to instantiate the skeletons and to start the skeleton activity.

A JaSkel skeleton is a Java class that extends the *Compute* class (Figure 1). The Skeleton method *eval* starts the skeleton activity.

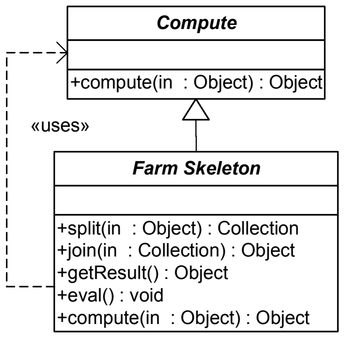

Compute
+compute(in : Object) : Object
«uses»
Farm Skeleton
+split(in : Object) : Collection
+join(in : Collection) : Object
+getResult() : Object
+eval() : void
+compute(in : Object) : Object

Fig. 1 The farm skeleton

Compute objects perform domain-specific computation. The programmer must create a subclass of class *Compute*, implementing the abstract method *Object compute(Object input)* to provide domain-specific computations.

To create a parallel application based on a farm parallel structure, the programmer must perform the following steps:

1. Create the worker *Compute* class.
2. Create the farmer class, extending *Farm*, implementing methods *split* and *join*.
3. Create instances of worker and farmer class to achieve the intended farm structure.
4. Start the skeleton activity, by calling the *eval* and grab the results with *getResult* method.

When the *eval* method is called on the farmer it will perform the following steps:

1. Split the initial data using the farmer *split* method.
2. Call *compute* method from workers with the pieces of data returned by method *split*;
3. Merge the partial results using the farmer *join* method.

JaSkel skeletons are also subclasses of *Compute* and implement the *compute* method (see Figure 1). This feature supports skeleton nesting, since a skeleton can be used in any context where a *Compute* instance can be used (e.g., in multi-level farms).

Figure 2 presents a simple pseudo-code for a two-level farm. Lines 01-06 define the *Worker* class, implementing the *compute* method. Lines 08-16 implement an *InnerFarm* class that creates its own set of *Worker* and provides specific *split* and *join* methods. Lines 18-27 implement the top-level farm, creating a set of inner farms. Lines 29-34 instantiate the *OuterFarm*, start the skeleton activity and get the computed task.

```
01  public class Worker extends Compute {
02     …
03     public Object compute(Object input) {
04        return /* processed input */;
05     }
06  }
07
08  public class InnerFarm extends Farm {
09     …
10     public InnerFarm() {
11        for(int i=0; i<myNumberOfInnerWorkers; i++)
12           /* … */ = new Worker();
13     }
14     Collection split(Object initialTask) { … }  // inner split
15     Object join(Collection partialResults) { … } // inner join
16  }
17
18  public class OuterFarm extends Farm {
19     …
20     public OuterFarm(Object task) {
21        super(task);       // save task into local structure
22        for(int i=0; i<myNumberOfOuterWorkers; i++)
23           /* … */ = new InnerFarm();
24     }
25     Collection split(Object initialTask) { … }  // outer split
26     Object join(Collection partialResults) { … } // outer join
27  }
28
29  public void main() {          // main function
30     Task task = … // task to compute
31     OuterFarm farmer = new OuterFarm(tasks);
32     farmer.eval();                // starts the farming process
33     Object o = farmer.getResult();       // get results
34  }
35
```

Fig. 2 Example of a two-level farm

JaSkel framework implements several types of farms: sequential, concurrent and dynamic. Dynamic farms are well suited for unbalanced problems and/or heterogeneous platforms, as it uses a demand-driven approach to distribute tasks among workers. Combining skeleton nesting with dynamic farms to achieve multi-level dynamic farms is particularly attractive to balance the load among several clusters and/or among cluster nodes. In the example of Figure 2 this would mean to simply extend the *DynamicFarm* instead of the sequential Farm (of course, the *split* method should also provide a number of tasks larger than the number of workers).

IV. Distribution Enabling Tools

JaSkel has built-in support for concurrency, through several concurrent skeletons. For instance, the *ConcurrentFarm* skeleton provides concurrent execution of *Compute* instances. These skeletons with built-in concurrency support can take advantage of shared memory machines, as they rely on a thread model which implicitly supports data sharing among threads. However, these skeletons can not take advantage of distributed memory machines as they rely on a shared address space.

Support for skeleton execution on distributed memory machines is provided by external tools. These tools transform concurrent skeletons into skeletons that can be executed in remote nodes. This approach has two main strong points: skeletons and distribution middleware can be composed in a more flexible manner and there is no need to use special hook points (e.g., to support remote creation, in [3], [7], skeletons must be created through object factories).

Using external tools to adapt skeletons for distributed systems also improve the development of the skeleton framework itself as the specificities of each middleware are addressed by these tools, making it easier to maintain and evolve the framework code and to adapt the framework for a new distribution middleware.

We have implemented two types of tools, both relying on source to source code transformations that generate AspectJ code [16], an extension to Java that supports aspect oriented programming [17]. Using AspectJ as the target language, instead of Java, allowed these tools to generate less code and to avoid changing the original skeleton code, enabling a more flexible composition among distribution tools. More details on using AspectJ to introduce distribution related code can be found in [18], [19].

The first tool generates Cluster-aware distributed skeletons based on Java RMI (more precisely UkaRMI [20], one of the most efficient Java RMI implementations today). The second tools converts a JaSkel skeleton to execute on computational Grids, by implementing *Compute* instances as jobs that are submitted to the resource manager of a local or remote clusters (we will simply call this the Grid enabling tool). The next two sub-sections present these tools in more detail. The last sub-section shows how these tools can be used together to deploy more Grid aware applications.

A. Cluster Enabling Tool

The purpose of the cluster enabling distribution tool is to support skeleton distribution among multiple JVMs residing on the same cluster or on a local area network. This tool transforms JaSkel *Compute* instances on Java RMI objects. These potentially remote objects may reside in another JVM. The generated code provides two base services: remote skeleton creation and remote invocation of *compute* methods (and other skeleton specific methods). The tool is based on a well known process [13], [21] that performs a source code transformation, based on 3 classes: proxy objects (PO), implementation objects (IO) and object managers (OM). A skeleton extending the *Compute* class is referred as the IO class and a generated aspect plays the rule of the PO class. Each node has an OM that implements local object factories to enable remote object creations. A similar strategy implemented through a bytecode rewriter is presented in [22]. However, the cluster enabling tool generates bytecodes by means of the AspectJ compiler.

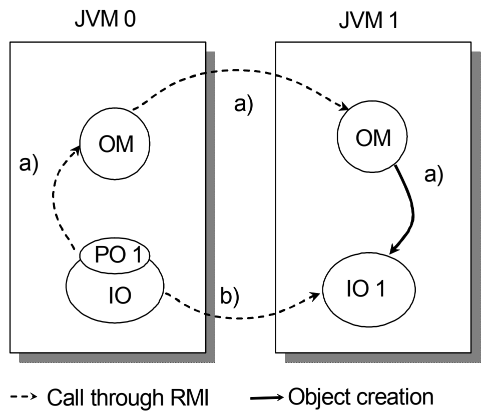


Fig. 3 Run-time system for RMI

Figure 3 illustrates the rules of PO, IO and OM objects in remote object creations and remote method calls. Whenever a *Compute* object was created in the original code the generated aspect (i.e., PO) requests an object creation to the local OM (JVM 0, call *a)* in the figure), which may locally create the IO object or forward the request to another OM. After remote object creation the PO transparently redirects local method calls to the remote IO (call *b)* in the figure).

```
public class Worker extends Compute {
     public Object compute(Object obj) {
          ... // method implementation
     }
}
```

Fig. 4 Sample Compute class

```
01  public interface IWorker {
02    public Object compute(Object) throws RemoteException;
03  }
04
05  public class ObjectManager {            // OM class
06       IWorker workerFactory() {
07            return( new Worker() );
08       }    …  // register Worker to be externally visible
09  }
10
11  aspect ProxyWorker {        // PO Aspect
12
13       declare parents: Worker implements IWorker;
14
15       IWorker myRemoteWorker; // reference to RMI stub
16
17       Worker around call (new Worker())  {
18                           // request remote object creation
19            IObjectManager ref2OM = ... // reference to OM
20            myRemoteWorker = refOM.workerFactory();
21            return( new PWorker() );  // fake local worker
22       }
23
24       Object around call (compute(Object obj)) {  // RMI
25            return (myRemoteWorker.compute(obj));
26       }
27  }
```

Fig. 5 RMI generated code for Compute class

Figure 4 presents a simple *Worker* skeleton that extends the *Compute* class and Figure 5 shows a simplified example of the AspectJ/Java RMI generated code for this class (to improve readability, exception handling code and minor AspectJ details are not included in the figure). The interface *IWorker* (lines 01-03) is created due to Java RMI requirements and the original worker class is declared to implement this interface (line 13, using the *declare parents* AspectJ construct). The *ObjectManager* is an object that implements a (remote) *Worker* factory (lines 05-09).

The *ProxyWorker* aspect transparently requests the remote *Worker* creation to the *ObjectManager* (lines 17-22) and redirects the *compute* calls for remote execution (lines 24-26). The *PWorker* class is created to implement a local proxy (code not shown), that extends the *Worker* class, to avoid the creation of multiple local farms in multi-level farms (line 21).

B. *Grid Enabling Tool*

The Grid enabling tool allows JaSkel skeletons to transparently run on computational Grids. This tool transforms JaSkel *Compute* instances into stand-alone applications that can run on a remote resource. These stand-alone applications gather the input data from a file that must be staged-in and write the result to another file that must staged-out to the client machine. Figure 6 shows a simplified example of the code generated by the Grid enabling tool for this purpose.

```
public static void main(String args[]) {
     Worker myWorker = new Worker();
     Object myData = … // read data from file
     Object result = myWorker.compute(myData);
     … // save result into output file
}
```

Fig. 6 Compute stand-alone application

Code generated for the client side share the structure of the generated code of the RMI tool, namely it also uses PO (e.g. Aspect) and IO objects. OMs are no longer explicitly required, since object instantiation is embedded into scripts that are submitted for execution (actually, OMs rule is replaced by resource discovery mechanisms to gather available clusters in Grid environments).

In the distribution-related code generated by the tool, POs and IOs interact in a different way: the PO transforms skeleton creation and method calls into scripts that are submitted to a local or a remote resource manager for execution, instead of directly performing these operations using the RMI middleware.

On the PO side, *Compute* object creations and *compute* method calls save the parameters into a specific file, generate a script that when executed creates the specific *Compute* object (i.e., IO) and calls its *compute* method. These *Compute* objects get their input data from the PO generated files and, after executing the *compute* method, save their results to files that are later retrieved by the PO (see Figure 6).

Figure 7 shows a simplified example of the code generated by this tool. The generated aspect (i.e., *ProxyWorker*) saves the *compute* parameter into a file, generates a script (line 09), submits this script for execution, waits for the creation of the file with the result data and returns this data.

```
01  aspect ProxyWorker {        // PO Aspect
02
03      Worker around call (new Worker())  {
04           return( new PWorker() );  // fake local worker
05      }
06
07      Object around call (compute(Object obj))
08           … // save obj into a specific file
09           … // generate script to execute remotely
10           submitJob(/* script name*/,/* list of files */);
11           while ( !myResultFile.exists() ) /* sleep */;
12           return(/* read data from file */);
13      }
14  }
```

Fig. 7 PO generated code for Compute class

To support multiple job submission systems the process of task submission is encapsulated into a procedure (line 10). Currently we provide implementations for PBS, SGE and GMarte [23]. Note that when the script is executed in a remote cluster the generated code must also provide commands to stage-in and stage-out the input/output files (this feature is automatically supported in these systems).

C. *Combining Distribution Tools*

Providing these distribution enabling tools as separate tools makes it possible a wide range of compositions:

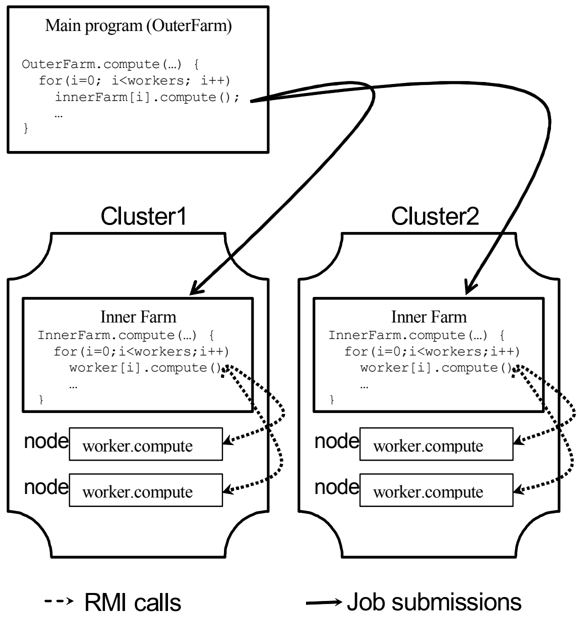
1. Base JaSkel skeletons supports local execution, taking advantage of multi-core machines;
2. Cluster enabling tool allows skeletons to run on a local cluster;
3. Grid enabling tools allows skeletons to run a generic Grid computational resources;
4. Combining distribution tools allows skeletons to run a computational Grid made of a federation of clusters.

Using both of the above tools in a single application is attractive in Grid environments to deploy parts of an application into different clusters and to use specific middleware among different elements of an application.

Combining these tools in a application composed of multi-level skeleton becomes most attractive. For instance, in a two level farm the first level can be distributed across the cluster nodes, using the cluster enabling tool, and the second level can take advantage of multi-processor nodes (i.e., by using no distribution middleware).

Combining the Grid enabling and the cluster enabling tools with two-level or three-level skeletons can be used to decompose an application into several parallel tasks that are submitted for execution into multiple clusters. JaSkel skeleton nesting, together with the injection of distribution code into skeletons (using the previously described tools) is a suitable way to develop this type of applications.

Figure 7 illustrates how the two-level farm from Figure 2 can be deployed on a computational Grid. Each *InnerFarm* is deployed on a remote cluster. This is accomplished by rewriting the *InnerFarm* and its *compute* method to generate and execute a job on a remote cluster, using the Grid enabling tool. Each *worker* belonging to an *InnerFarm* is deployed in a node of the local cluster. In this *InnerFarm* communication between the farmer and the workers uses the RMI communication middleware. Communication between the *OuterFarm* and the *InnerFarm* is performed through files that are automatically generated by the Grid tool.



```
OuterFarm.compute(…) {
  for(i=0; i<workers; i++)
    innerFarm[i].compute();
    …
}
```

```
InnerFarm.compute(…) {
  for(i=0;i<workers;i++)
    worker[i].compute();
    …
}
```

Fig. 8 Deployment of multi-level skeletons

This example can be extended with a third skeleton level to take advantage of multi-core machines. Technically the approach can use n-level skeletons and multiple distribution tools.

Skeletons and skeleton nesting are particularly useful to inject distribution code by means of external tools. Since skeletons follow well defined rules (see Figure 1) it is feasible to provide tools to inject this code into skeletons (e.g., by rewriting the skeleton implementation and the *compute* method). Moreover, aspect-oriented techniques make more manageable this type of approach as the original classes do not need to be rewritten as in traditional approaches [13].

V. Performance Evaluation

The evaluation presented here aims to show that, by providing external tools to adapt JaSkel skeletons to specific communication middleware, JaSkel applications can easily scale to a wide range of platforms, from multi-core systems to computational Grids. We also aim to show that by combining these distribution tools with multi-level skeletons makes it feasible to develop applications that can take advantage of the hierarchical nature of computational Grids. The important point is that all these results were obtained with the same JaSkel application (i.e., code base), using a single or a multi-level farm and by injecting distribution code through our tools.

Our first benchmark shows the execution time of JaSkel skeletons using the tools presented in the previous section. We present execution times for a reference algorithm: a parallel ray tracer (image size of 4000), from the Java Grande Forum [24]. These results were collected on a simulated Grid environment, consisting of two clusters with 4 nodes each (each node is a dual Xeon 3.2 GHz 2MB L2 with 2GB RAM). Presented values are the median of 10 runs.

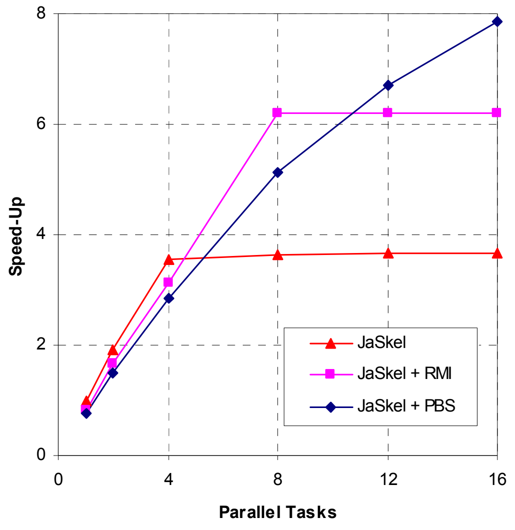


Fig. 9 RayTracer speed-up for different communication middleware

Figure 8 shows the speed-up relative to the sequential JGF RayTracer (on a Xeon 5130 2.0 GHz) of several JaSkel farm implementations: the built-in concurrent farm on a quad-core machine (*JaSkel*), the farm pre-processed by the Cluster enabling tool (*JaSkel + RMI*), which run on a simple cluster and the farm pre-processed by the Grid enabling tool (*JaSkel + PBS*) which run on these two clusters. The built-in JaSkel support for concurrent farm only scales up to 4 CPU, as it can not distribute *Compute* skeletons through cluster nodes. The RMI version only scales within one cluster (up to 8 parallel tasks). The PBS version has the advantage of supporting skeletons that may run on multiple clusters, as it can submit jobs to several clusters (e.g., computational Grids), however, both the PBS and the RMI versions impose additional overheads on a small number of nodes, since the cluster nodes are slower than the 4-core machine. Each version also adds some extra overheads. In the RMI case due to the less efficient communications, when compared with a shared-memory implementation. In the PBS case, an additional overhead is due to job submission (including stage in and stage out of input/output files).

TABLE I
JASKEL VS NATIVE (JGF) EXECUTION TIMES

	Multi-Thread		RMI	
	JGF	**JaSkel**	**JGF**	**JaSkel**
1	22.0	21.8	28.1	28.3
2	11.2	11.5	14.4	14.5
4	06.2	06.3	07.6	07.6
8	-	-	04.3	04.3

The overhead of these implementations, relative to native implementations, is very low. Table 1 compares the execution times of the RayTracer application (500x500 image) against the JGF Multithread benchmark and against an equivalent RMI implementation.

The second experiment evaluates the use of multiple clusters and multi-level farms to take advantage of intra-cluster/intra-node communication

Each cluster has a different front-end which conceptually simulates two different organisations (with access provided only through *ssh* and *scp*). These two clusters are in the same network which strongly reduces the cost of inter-cluster communication (e.g., file copy and job submission). To further simulate a real environment, by increasing the impact of these costs, we also scaled down the RayTracer image size in this case to 500.

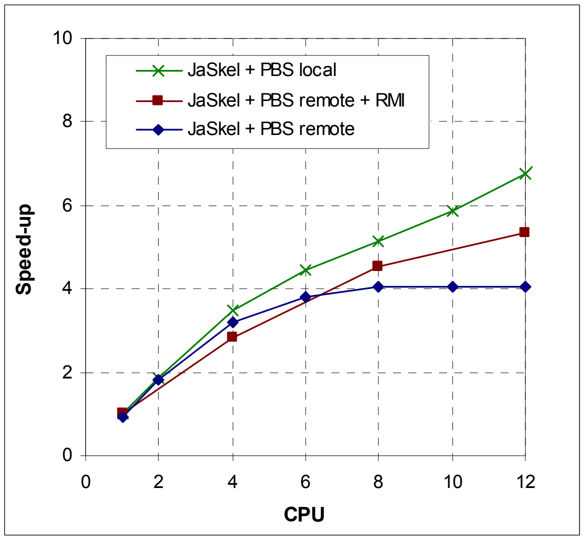


Fig. 10 RayTracer speed-ups in multi-clusters

Figure 9 presents the speed-ups obtained when running on single cluster, locally submitting all tasks (*JaSkel PBS local*). We also present results when tasks are remotely submitted into both clusters (*JaSkel PBS remote*) and when we remotely submit parallel RMI based tasks, using a two level farm (*JaSkel PBS remote + RMI*). In all these versions the PBS tool was used to generate the required code to execute JaSkel skeletons on several nodes/clusters. On the multi-level farm (with code very close to the provided in Figure 2) we deployed an inner farm on each cluster, consisting of half of the total workers (e.g., there are always two inner farms, each using half of the CPU) that are scheduled for execution into the same cluster. Calls to the farmer of the inner farm are converted into tasks that are submitted to the PBS queues.

When tasks are remotely submitted on a small number of CPU (e.g., 4), the multilevel farm introduces a small overhead due to the need to create an additional farm level (including additional work partition and communication). However, when using a larger number of CPUs the use of RMI to locally communicate within an inner farm overcomes this small overhead by reducing the overall number of tasks submitted for execution. A two level farm has fewer remote tasks, leading to lower communication costs, which makes this version attractive in computational Grids built with several clusters of multi-core and multiprocessor machines.

These results suggest that multi-level farms and the use of independent tools to inject distribution related code into applications may, more efficiently, take advantage of Grids of clusters.

Skeleton approaches present specific benefits in this path, as they inherently support skeleton nesting (e.g., hierarchically composed skeletons). Moreover, applications built with skeletons follow more well defined rules (for instance, by extending a common base class), which makes it easier to inject distribution related code into applications in a transparent manner. Skeleton nesting, together with the injection of distribution code into selected skeletons is an effective way to deal with different interconnection middleware providing different latencies and bandwidths, which is one of the main difficulties when developing parallel applications targeted for a Grid of clusters environment.

VI. Conclusions

JaSkel offers a skeleton based approach to structure parallel applications. The skeleton framework aims to improve programmer's productivity and, at the same time, to provide code that efficiently execute on a wide range of platforms. Skeleton based approaches that support skeleton nesting are particularly well suited for computational Grids. In JaSkel it is possible to develop applications that closely match the hierarchical nature of computational Grids. This communication showed that JaSkel applications can scale to a wide range of platforms and how skeleton nesting can improve the performance of Grid applications.

JaSkel uses a new approach to inject distribution code into skeleton based applications. It relies on external tools, providing a mean to fine-tune applications to the environment where they run. This feature also helps to support multiple levels of parallelism within an application.

Current work includes the development of a more sophisticated runtime environment that provides adaptation of skeletons to particularly running conditions (e.g., available compute nodes). Particularly interesting is how multi-level skeletons can interplay with Grid meta-schedulers, such as the GridWay.

Acknowledgment

This work was supported by PPC-VM project (Portable Parallel Computing Based on Virtual Machines, POSI/CHS/47158/2002) and AspectGrid project (Pluggable Grid Aspects for Scientific Applications, GRID/GRI/81880/2006), funded by Portuguese FCT and European funds (FEDER). Performance results where collected in the SeARCH cluster (Services & Advanced Computing with HTC/HPC, CONC-REEQ/443/EEI/2005), also funded by Portuguese FCT and European funds (FEDER).

References

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

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

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

[4] 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.

[5] J. Fernando, J. Sobral, A. Proença. "JaSkel: A Java Skeleton-Based Framework for Structured Cluster and Grid Computing", *IEEE CCGrid'06*, Singapore, May 2006.

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

[7] J. Gorlatch, J. Dunnweber, J. "From Grid Middleware to Grid Applications: Bridging the Gap with HOCs, *Future Generation Grids*, Springer, 2006.

[8] M. Cole, "Bringing skeletons out of the closet: a pragmatic manifesto for skeletal parallel programming", *Parallel Computing*, Vol. 30 , n. 3, March 2004.

[9] Y. Aridor, M. Factor, A. Teperman, "cJVM: A Single System Image of a JVM on a Cluster", *International Conference on Parallel Processing*, Wakamatsu, Japan, September 1999.

[10] G. Antoniu, L. Bougé, P. Hatcher. , M. MacBeth, K. McGuigan, R. Namyst, "The hyperion system: Compiling multi-threaded java bytecode for distributed execution", *Parallel Computing*, vol. 27, no. 10, September 2001.

[11] R. Veldema, R. Bhoedjang, H. Bal, "Jackal, a compiler based implementation of java for clusters of workstations", *ACM PPoPP'01*, Utah, USA, June 2001.

[12] W. Zhu, C. Wang, F. Lau, "JESSICA2: Distributed Java Virtual Machine with Transparent Thread Migration Support", *IEEE Cluster 2002*, Chicago, USA, September 2002.

[13] M. Philippsen, M. Zenger, "JavaParty – transparent remote objects in Java", *Concurrency: Practice and Experience*, vol.19 n.11, November 1997.

[14] F. Baude, L. Baduel, D. Caromel, A. Contes, F. Huet, M. Morel R. Quilici, "Programming, Composing, Deploying for the Grid", *GRID COMPUTING: Software Environments and Tools*, Jose C. Cunha and Omer F. Rana (Eds), Springer Verlag, January 2006.

[15] K. Reeuwijk, R. Niewpoort, H. Bal, "Developing Java Grid applications with Ibis", *11th International Euro-Par Conference*, Lisbon, Portugal, September 2005.

[16] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, W. Griswold, "An Overview of AspectJ", *ECOOP 2001*, June 2001.

[17] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. Loingtier, J. Irwin, "Aspect Oriented Programming", *ECOOP'97*, June 1997.

[18] S. Soares, L. Loureiro, P. Borba, "Implementing Distribution and Persistence Aspects With AspectJ", *OOPSLA '02*, November 2002.

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

[20] M. Philippsen, B. Haumacher, C. Nester, "More Efficient Serialization and RMI for Java", *Concurrency: Practice and Experience*, vol. 12 n. 7, May 2000.

[21] J. Sobral, A. Proença, "A Run-time System for Dynamic Grain Packing", *Euro-Par'99*, Toulouse, France, September 1999, LNCS vol. 1685, Springer 1999.

[22] M. Factor, A. Schuster, K. Shagin, "JavaSplit: a runtime for execution of monolithic Java programs on heterogenous collections of commodity workstations, *IEEE Cluster*, Hong Kong, December 2003.

[23] J. M. Alonso, V. Hernández, G. Moltó, "GMarte: Grid Middleware to Abstract Remote Task Execution", *Concurrency and Computation: Practice and Experience*, 18(15), 2006.

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