Using Skeletons in a Java-Based Grid System

Martin Alt and Sergei Gorlatch

Technische Universität Berlin, Germany
{mnalt|gorlatch}@cs.tu-berlin.de

Abstract. Grid systems connect high-performance servers via the Internet and make them available to application programmers. This paper addresses the challenge of software development for Grids, by means of reusable algorithmic patterns called *skeletons*. Skeletons are generic program components, which are customizable for a particular application and can be executed remotely on high-performance Grid servers. We present an exemplary repository of skeletons and show how a particular application, FFT (Fast Fourier Transform), can be expressed using skeletons and then executed using RMI (Remote Method Invocation). We describe a prototypical Java-based Grid system, present its optimized RMI mechanism, and report experimental results for the FFT example.

1 Introduction

Grid systems connect high-performance computational servers via the Internet and make them available to application programmers. While the enabling infrastructures for Grid computing are fairly well developed [1], initial experience has shown that entirely new approaches are required for Grid programming [2]. A particular challenge is the phase of algorithm design: since the type and configuration of the servers on which the program will be executed is not known in advance, it is difficult to make the right design decisions, to perform program optimizations and estimate their impact on performance.

We propose to address Grid programming by providing the application programmers with two kinds of software components on the server side: (1) traditional library functions, and (2) reusable, high-level patterns, called *skeletons*. Skeletons are generic algorithmic components, customizable for particular applications by means of their functional parameters. Time-intensive skeleton calls are executed remotely on high-performance Grid servers, where architecture-tuned, efficient parallel implementations of the skeletons are provided.

The contributions and organization of the paper are as follows: We present the structure of our Grid system based on Java and RMI and explain the advantages of skeletons on the Grid (Sect. 2). We introduce a repository of skeletons for expressing parallel and distributed aspects of Grid applications (Sect. 2.1), discuss the inefficiency of standard Java RMI on the Grid and propose using future-based RMI (Sect. 2.2). We show how a mathematical specification of FFT is expressed using our skeletons, develop a skeleton-based Java program and report experimental performance results for it (Sect. 3). We conclude by discussing our results in the context of related work.

© Springer-Verlag Berlin Heidelberg 2003

2 Our Grid Prototype

Our prototypical Grid environment (Fig. 1) consists of two university LANs – one at the Technical University of Berlin and the other at the University of Erlangen. They are connected by the German academic internet backbone (WiN), covering a distance of approx. 500 km. There are three high-performance servers in our Grid: a shared-memory multiprocessor SunFire 6800, a MIMD supercomputer of type Cray T3E, and a Linux cluster with SCI network. Application programmers work from clients (PCs and workstations). A central entity called "lookup service" is used for resource discovery. The reader is referred to [3] for details of the system architecture and the issues of resource discovery and management.

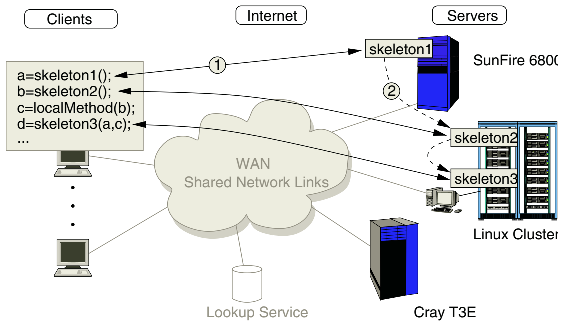

Fig. 1. System architecture and interaction of its parts

We propose developing application programs for such Grid systems using a set of reusable, generic components, called *skeletons*. As shown in the figure, a program on a client is expressed as a sequential composition of skeleton calls and local calls. The servers in the Grid provide architecture-tuned implementations of the skeletons: multithreaded, MPI-based, etc. Applications composed of skeletons can thus be assigned for execution to particular servers in the Grid with a view to achieving better performance.

Time-intensive skeleton calls are executed remotely on servers which provide implementations for the corresponding skeleton (arrow ①in the figure). If two subsequent skeleton calls are executed on different servers, then the result of the first call must be communicated as one of inputs for the second call (arrow ②). This situation is called *composition of skeletons*.

Using skeletons for programming on the Grid has the following advantages:

– Skeletons' implementations on the server side are usually highly efficient because they can be carefully tuned to the particular server architecture.
– The once-developed, provably correct implementation of a skeleton on a particular server can be reused by different applications.
– Skeletons hide the details about the executing hardware and the server's communication topology from the application programmer.
– Skeletons provide a reliable model for performance prediction, providing a sound information base for selecting servers.

2.1 A Repository of Skeletons

In the following, we describe a (by no means exhaustive) collection of skeletons. Since at least some of the skeletons' parameters are functions, skeletons can be formally viewed as higher-order functions. In practice, functional parameters are provided as program codes, in our system as Java bytecodes.

We begin our presentation with simple skeletons that express data parallelism:

Map: Apply a unary function f to all elements of a list:
$map(f, [x_1, \ldots, x_n]) \; = \; [f(x_1), \ldots, f(x_n)]$

Scan: Compute prefix sums of a list by traversing the list from left to right and applying a binary associative operator $\oplus$:
$scan(\oplus, [x_1, \ldots, x_n]) \; = \; [\, x_1, \, (x_1 \oplus x_2), \ldots, (\cdots(x_1 \oplus x_2) \oplus x_3) \oplus \cdots \oplus x_n) \,]$

A more complex data-parallel skeleton, DH (Distributable Homomorphism) [4], expresses a divide-and-conquer pattern with parameter operators $\oplus$ and $\otimes$:

DH: Formally, $dh(\oplus, \otimes, x)$ transforms a list $x = [x_1, \ldots, x_n]$ of size 2^l into a result list $y = [y_1, \ldots, y_n]$, whose elements are computed as follows:

$$y_i = \begin{cases} u_i \oplus v_i, & \text{if } i \leq n/2 \\ u_{i-n/2} \otimes v_{i-n/2}, & \text{otherwise} \end{cases} \tag{1}$$

where $u = dh(\oplus, \otimes, [x_1, \ldots, x_{n/2}])$ and $v = dh(\oplus, \otimes, [x_{n/2+1}, \ldots, x_n])$.

In addition to these data-parallel skeletons, we provide two auxiliary skeletons, whose aim is efficient communication between client and server:

Replicate: Create a new list containing n times element x: $repl(x, n) = [x, \ldots, x]$. The *repl* skeleton can be called remotely on a server to create there a list of n identical elements, instead of sending the whole list over the network.

Apply: Applies a unary function f to a parameter x: $apply(f, x) = f(x)$. The *apply* skeleton is used to remotely execute a function f by shipping its code to the server, rather than moving the data to the client, executing the function locally and then sending the result to the server again.

Our skeleton-based Grid programmimg environment for the system shown in Fig. 1 is built on top of Java and RMI. We chose the Java platform mostly for reasons of portability (see [5] for "10 reasons to use Java in Grid computing").

In the system, skeletons are offered as Java (remote) interfaces, which can be implemented in different ways on different servers. To be as flexible as possible, all skeletons operate on `Objects` or arrays of `Object`. For example, the interface for the *scan* skeleton contains a single method

```
public Object[] invoke(Object[], BinOp oplus);
```

To use the `scan` skeleton, the client first finds a server for execution, using the lookup service (see [3] for details). After obtaining an RMI reference to the `scan` implementation on the server, the skeleton is executed via RMI by calling the `invoke` method with appropriate parameters.

2.2 Future-Based RMI for the Grid

Using the RMI mechanism in Grid programs has the important advantage that the outsourcing of skeleton calls to remote servers is transparent for the programmer: remote calls are coded in exactly the same way as local calls. However, since the RMI mechanism was developed for client-server systems, it is not optimal for the Grid. We illustrate this using the following example: a composition of two skeleton calls, with the result of the first call being used as an argument of the second call (`skeleton1` and `skeleton2` are remote references):

```
result1 = skeleton1.invoke(...);
result2 = skeleton2.invoke(result1,...);
```

Executing such a composition of methods using standard RMI, the result of a remote method invocation is always sent back directly to the client. This is exemplified for the above example in Fig. 2 (left). When `skeleton1` is invoked (①), the result is sent back to the client (②), then to `skeleton2` (③). Finally, the result is sent back to the client (④). For typical applications consisting of many composed skeletons, this feature of RMI results in very high time overhead.

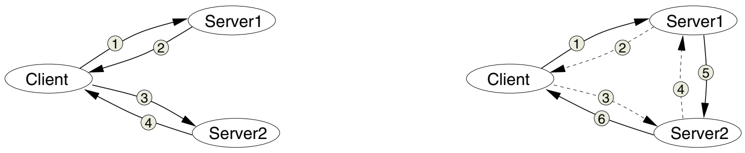

Fig. 2. Skeleton composition using plain RMI (left) and future-based RMI (right)

To eliminate this overhead, we have developed so-called *future-based RMI*: an invocation of a skeleton on a server initiates the skeleton's execution and then returns immediately, without waiting for the skeleton's completion (see Fig. 2, right). As a result of the skeleton call, a future reference is returned to the client (②) and can be used as a parameter for invoking the next skeleton (③). When the future reference is dereferenced (④), the dereferencing thread on the server is blocked until the result is available, i. e. the first skeleton actually completes. The result is then sent directly to the server dereferencing the future reference (⑤). After completion of `skeleton2`, the result is sent to the client (⑥).

Compared with plain RMI, our future-based mechanism substantially reduces the amount of data sent over the network, because only a reference to the data is sent to the client; the result itself is communicated directly between the servers. Moreover, communications and computations overlap, effectively hiding latencies of remote calls. We have implemented future-based RMI on top of SUN Java RMI and report experimental results in Sect. 3.3. (see [6] for further details).

Future references are available to the user through a special Java interface `RemoteReference`. There are only few differences when using future-based RMI compared with the use of plain RMI: (1) instead of `Objects`, all skeletons return values of type `RemoteReference`, and (2) skeletons' interfaces are extended by `invoke` methods, accepting `RemoteReferences` as parameters.

3 Case Study: Programming FFT Using Skeletons

By way of an example application, we consider the Fast Fourier Transformation (FFT). The FFT of a list $x = [x_0, \ldots, x_{n-1}]$ of length $n = 2^l$ yields a list whose i-th element is defined as $(\mathrm{FFT}\, x)_i = \sum_{k=0}^{n-1} x_k \omega_n^{ki}$, where ω_n denotes the n-th complex root of unity, i. e. $\omega_n = \mathrm{e}^{2\pi\sqrt{-1}/n}$.

3.1 Expressing FFT with Skeletons

We now outline how the FFT can be expressed as a composition of skeletons (see [4] for details). The FFT can be written in divide-and-conquer form as follows, where $u = [x_0, x_2, \ldots, x_{n-2}]$ and $v = [x_1, x_3, \ldots, x_{n-1}]$:

$$(\mathrm{FFT}x)_i = \begin{cases} (\mathrm{FFT}u)_i \;\hat{\oplus}_{i,n}\; (\mathrm{FFT}v)_i & \text{if } i < n/2 \\ (\mathrm{FFT}u)_{i-n/2} \;\hat{\otimes}_{i-n/2,n}\; (\mathrm{FFT}v)_{i-n/2} & \text{else} \end{cases} \tag{2}$$

where $a \,\hat{\oplus}_{j,m}\, b = a + \omega_m^j b$, and $a \,\hat{\otimes}_{j,m}\, b = a - \omega_m^j b$.

The formulation (2) is close to the *dh* skeleton format from Sect. 2.1, except for $\hat{\oplus}$ and $\hat{\otimes}$ being parameterized with the position i of the list element and the length n of the input list. Therefore we express the FFT as instance of the *dh* skeleton, applied to a list of triples (x_i, i, n), with operator $\oplus$ defined on triples as $(x_1, i_1, n_1) \oplus (x_2, i_2, n_2) = (x_1 \,\hat{\oplus}_{i_1,n_1}\, x_2, i_1, 2n_1)$. Operator $\otimes$ is defined similarly.

Computing FFT using skeletons: As skeletons are higher-order functions, we first provide a functional program for FFT, which is then transformed to Java in a straightforward manner. The FFT function on an input list x can be expressed using skeletons by transforming the input list into a list of triples, applying the *dh* skeleton and finally taking the first elements of the triples for the result list:

$$\mathrm{FFT} \;=\; \mathit{map}(\pi_1) \,\circ\, \mathit{dh}\,(\oplus, \otimes) \,\circ\, \mathit{apply}(\mathit{triple})$$

where *triple* is a user-defined function that transforms a list $[x_1, \ldots, x_n]$ to the list of triples $[(x_1, 1, 1), \ldots, (x_i, i, 1), \ldots, (x_n, n, 1)]$, and $\circ$ denotes function composition from right to left, i. e. $(f \circ g)\ (x) \;=\; f(g(x))$.

Both operators $\hat{\oplus}$ and $\hat{\otimes}$ in (2) repeatedly compute the roots of unity ω_n^i. Instead of computing these for every call, they can be computed once a priori and stored in a list $\Omega = [\omega_n^1, \ldots, \omega_n^{n/2}]$, accessible by both operators, thus reducing computations. Using the relation $\omega_m = \omega_n^{n/m}$, the computation of ω_m^i in $\hat{\oplus}/\hat{\otimes}$ can be replaced with $\pi(ni/m, \Omega)$, where $\pi(k, \Omega)$ selects the k-th entry of Ω. Therefore, $\hat{\oplus}$ can be expressed as $a \,\hat{\oplus}_{j,m,\Omega}\, b = a + \pi(nj/m, \Omega) b$. Operator $\hat{\otimes}$ can be expressed using Ω analogously. Thus, $\oplus/\otimes$ are parameterized with Ω; we express this by writing $\oplus(\Omega)/\otimes(\Omega)$ in the following.

Now, we can express the computation of Ω using the *repl* and the *scan* skeletons, and arrive at the following skeleton-based program for the FFT:

$$\begin{aligned} \Omega &= \mathit{scan}(*) \circ \mathit{repl}(n/2, \omega_n) \\ \mathrm{FFT} &= \mathit{map}(\pi_1) \,\circ\, \mathit{dh}\,(\oplus(\Omega), \otimes(\Omega)) \,\circ\, \mathit{apply}(\mathit{triple}) \end{aligned} \tag{3}$$

where $*$ denotes complex number multiplication.

3.2 Skeleton-Based Java Program for FFT

The Java code for the FFT, obtained straightforwardly from (3), is as follows:

```
  //repl,scan,map,dh are remote refs to skeletons on servers
  //compute roots of unity
RemoteReference r = repl.invoke(length, omega_n);
RemoteReference omegas = scan.invoke(r, new ScanOp());
  //instantiate operators for dh
oplus = new FFTOplus(omegas);
otimes = new FFTOtimes(omegas);
  //fft
r = apply.invoke(inputList, new TripleOp());
r = dh.invoke(oplus, otimes, r);
r = map.invoke(r, new projTriple());
  //get result
result = r.getValue();
```

At first, the roots of unity are computed, using the *repl* and *scan* skeletons. Both `repl` and `scan` are RMI references to the skeletons' implementation on a remote server, obtained from the lookup service. Execution of the skeletons is started using the `invoke` methods. Variable `OmegaN` passed to the `repl` skeleton corresponds to ω_n and `omegas` corresponds to Ω. As a binary operator for `scan`, complex multiplication is used, implemented in class `ComplexMult`. The operators $\oplus(\Omega)$ and $\otimes(\Omega)$ for the *dh*-skeleton are instantiated as objects of classes `FFTOplus` and `FFTOtimes` on the client side. The constructor for the parameters receives the list Ω as an argument. Each operator stores a reference to the list in a private variable in order to access it later for computations.

Next, the FFT itself is computed in three steps. First the input list is transformed to a list of triples, using the `apply` skeleton with a user-defined function. Then the `dh`-skeleton is called on the list of triples, using the two customizing operators defined earlier. Finally, the list of result values is retrieved from the list of triples using the `map` skeleton with an instance of the user-defined class `projTriple`.

In a preliminary step (omitted in the code presented above), the program obtains from the lookup service a remote reference for each skeleton used in the program (`repl`, `scan`, `map` and `dh`). The program is executed on the client side; all calls to the `invoke` method of the involved skeletons are executed remotely on servers via RMI.

3.3 Experimental Results

We measured the performance of the skeleton-based FFT program using the testbed of Fig. 1. We used a SunFire 6800 with 12 US-III+ 900 MHz processors in Berlin as our server and an UltraSPARC-IIi 360 MHz as client in Erlangen, both using SUN's JDK1.4.1 (HotSpot Client VM in mixed mode). Because there were several other applications running on the server machine during our experiments, a maximum of 8 processors was available for measurements.

Fig. 3 shows the runtimes for different problem sizes (ranging from 2^{15} to 2^{18}) and four different versions of the program: the first running locally on the client ("local FFT"), second using plain RMI, third version using future-based RMI, and the fourth version where the FFT is executed as a single server sided method called from the client ("ideal remote"). We consider the fourth version as ideal, as there is no overhead for remote composition of skeletons for that version: it corresponds to copying the whole program to the server and executing it there. For the plain RMI version, only the *scan* and *dh* skeletons are executed on the server, because all parameters and results are transmitted between client and server for each method call using plain RMI, so that executing the *repl*, *apply* and *map* skeleton remotely would slow down the program unnecessarily. For the future-based RMI version, all skeletons are executed on the server.

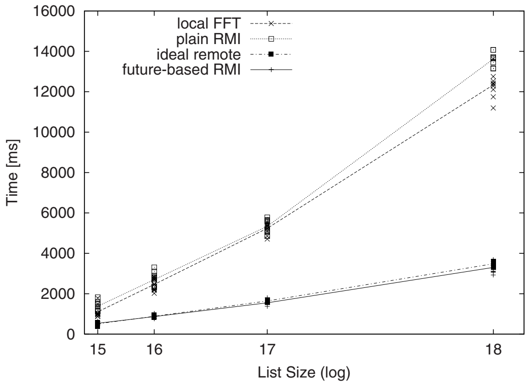

Fig. 3. Measured runtimes for the FFT programs

The figure shows ten measurements for each program version, with the average runtimes for each parameter size connected by lines. The plain RMI version is much (three to four times) slower than the future-based RMI version and unable to outperform the local, client sided FFT. Thus, the communication overhead outweighs the performance gain for execution on the server.

By contrast, the future-based RMI version eliminates most of the overhead and is three to four times faster than the local version. Compared with the "ideal remote" case the runtimes are almost identical. For large input lists (2^{17} and 2^{18}), the future-based version is even slightly faster than the remote version. This is due to the fact, that the future-based version invokes skeletons asynchronously, so the *apply* skeleton is already called while the *scan* skeleton is still running. Thus, using future-based RMI allows an efficient execution of programs with compositions of remote methods, in particular compositions of skeletons.

4 Conclusions and Related Work

In this paper, we have addressed the challenging problem of software design for heterogeneous Grids, using a repository of reusable algorithmic patterns called *skeletons*, that are executed remotely on high-performance Grid servers. While the use of skeletons in the parallel setting is an active research area, their application for the Grid is a new, intriguing problem.

We have described our prototypical Grid system. Java and RMI were chosen to implement our system in order to obtain a highly portable solution. Other promising opportunities include, e. g. the Lithium system [7] for executing mainly task-parallel skeletons in Java. We have proposed a novel, future-based RMI mechanism, which substantially reduces communication overhead for compositions of skeleton calls. It differs from comparable approaches because it combines hiding network latencies using asynchronous methods (as in [8,9]) and reducing network dataflow by allowing server/server communication (e. g. found in [10]).

We have proposed an exemplary (and by no means exhaustive) repository of skeletons, which includes several elementary data-parallel functions, the divide-and-conquer skeleton DH, and two auxiliary skeletons which are helpful in a Grid environment. We have demonstrated how a mathematical description of FFT (Fast Fourier Transform) can be expressed using our skeletons, leading to an efficient Java program with remote calls for skeletons.

At present, each skeleton call is executed on a single Grid node. We plan to allow distribution of skeletons across several nodes in the future, at least for task-parallel and simple data-parallel skeletons.

References

1. Foster, I., Kesselmann, C., eds.: The Grid: Blueprint for a New Computing Infrastructure. Morgan Kaufmann (1998)
2. Kennedy, K., et al.: Toward a framework for preparing and executing adaptive grid programs. In: Proc. of NSF Next Generation Systems Program Workshop (2002)
3. Alt, M., et al.: Algorithm design and performance prediction in a Java-based Grid system with skeletons. In: Proc. of Euro-Par 2002. LNCS Vol. 2400, Springer (2002)
4. Gorlatch, S., Bischof, H.: A generic MPI implementation for a data-parallel skeleton: Formal derivation and application to FFT. Parallel Processing Letters **8** (1998)
5. Getov, V., et al.: Multiparadigm communications in Java for Grid computing. Communications of the ACM **44** (2001) 118–125
6. Alt, M., Gorlatch, S.: Optimizing the use of Java RMI for Grid application programming. Technical Report 2003/08, TU Berlin (2003) ISSN 1436-9915.
7. Danelutto, M., Teti, P.: Lithium: A structured parallel programming enviroment in Java. In: Proc. of Computational Science. ICCS. LNCS Vol. 2330, Springer (2002)
8. Raje, R., Williams, J., Boyles, M.: An asynchronous remote method invocation (ARMI) mechanism in Java. Concurrency: Practice and Experience **9** (1997)
9. Falkner, K.K., Coddington, P., Oudshoorn, M.: Implementing asynchronous remote method invocation in Java. In: Proc. of Parallel and Real Time Systems (1999)
10. Yeung, K.C., Kelly, P.H.J.: Optimising Java RMI programs by communication restructuring. In: Middleware 2003, Springer (2003)