Algorithm Design and Performance Prediction in a Java-Based Grid System with Skeletons

Martin Alt, Holger Bischof, and Sergei Gorlatch

Technical University of Berlin, Germany

Abstract. We address the challenging problem of algorithm design for the Grid by providing the application user with a set of high-level, parameterized components called *skeletons*. We describe a Java-based Grid programming system in which algorithms are composed of skeletons. The advantage of our approach is that skeletons are reusable for different applications and that skeletons' implementations can be tuned to particular machines of the Grid with quite well-predictable performance.

1 Introduction

One of the main challenges in application programming for the Grid is the phase of algorithm design and, in particular, performance prediction early on in the design process: it is difficult to choose the right algorithmic structure and perform architecture-specific optimizations of an application, because the type of machine the program will actually be executed on is not known in advance.

We propose providing Grid application programmers with a set of high-level algorithmic patterns, called *skeletons*. Skeletons are used as program components, customizable for particular applications. Computational servers of the Grid provide possibly different, architecture-dependent implementations of the skeletons, which can be tuned for execution on particular Grid servers.

The advantage of our approach is that applications can be conveniently expressed using reusable algorithmic skeletons, for which reliable performance estimates on a particular Grid server are available. This facilitates systematic rather than *ad hoc* design decisions on both the algorithmic structure of an application and the assignment of application parts to servers.

In this paper, we describe an experimental Java-based programming system with skeletons for a Grid environment, with focus on the critical problem of performance prediction in the course of algorithm design. The particular contributions and the structure of the paper are as follows:

- An architecture of the Grid programming system is proposed, in which the user chooses a suitable server for each skeleton invocation in his application program (Section 2).
- A Java+RMI experimental implementation is presented, with Java bytecodes used as application-specific parameters of skeletons (Section 3).
- A simple performance model for remote execution of skeletons on the Grid servers is proposed and tested using system measurements. (Section 4).

We conclude the paper by discussing our findings in the context of related work.

B. Monien and R. Feldmann (Eds.): Euro-Par 2002, LNCS 2400, pp. 899–906.

2 System Architecture and Skeletons

The idea of Grid programming with skeletons is to separate two phases of programming – algorithm design and implementation. The user composes his program using predefined algorithmic patterns (skeletons), which appear as function calls with application-specific parameters. The actual organization of parallelism is left to the skeleton implementation, which is provided on the server side and is geared to a particular architecture of a Grid server, e. g. distributed- or shared-memory, multithreaded, etc.

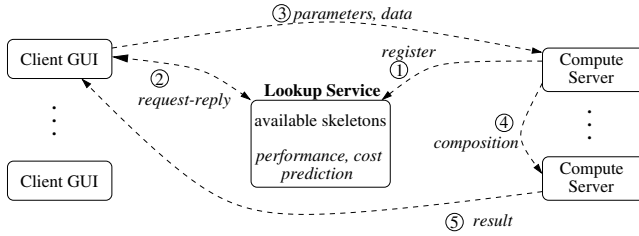


Fig. 1. System architecture and interaction of its parts

We propose the following system architecture, consisting of three kinds of components: user machines (*clients*), target machines (*servers*) and the central entity, called *lookup service* (see Fig. 1).

Each server provides a set of skeletons that can be invoked from the clients. Invoking skeletons remotely involves the following steps, also shown in Figure 1:

① Registration: Each server registers the skeletons it provides with the lookup service to make them accessible to clients. Together with each skeleton, a performance estimation function is registered, as explained below.
② Service request-reply: A client queries the lookup service for a skeleton it needs for an application and is given a list of servers implementing the skeleton. The skeletons that will actually be used are selected (using heuristics or tool-driven by the user).
③ Skeleton invocation: During the program execution, skeletons are invoked remotely with application-specific parameters.
④ Composition: If the application consists of a composition of skeletons, they may all be executed either on the same server or, alternatively, in a pipelined manner across several servers.
⑤ Skeleton completion: When the compute server has completed the invoked skeleton, the result is sent back to the client.

Skeleton Examples. In this paper, we confine our attention to so-called *data-parallel* skeletons whose parallelism stems from partitioning the data among processors and performing computations simultaneously on different data chunks.

For the sake of simplicity, our running example in the paper is an elementary data-parallel skeleton called *reduction*: function $reduce(\oplus)$ computes the "sum" of all elements in a data structure using the associative customizing operator $\oplus$. For example, for a list of three elements $[a, b, c]$, the result is $a \oplus b \oplus c$.

Formally, *reduce* is a *higher-order function* whose argument is the customizing operator $\oplus$. Implementations of *reduce* on particular servers are parameterized with $\oplus$ as well; they expect the operator to be provided during invocation. Note that the customizing operator $\oplus$ itself may be time-consuming: e. g. below we consider reduction on a list of matrices, where the operator denotes matrix multiplication. Another, quite similar example is the scan skeleton, $scan(\oplus)$, which computes the prefix sums using an associative operator $\oplus$, i. e. applying $scan(\oplus)$ to list $[a, b, c]$ would result in $[a, a \oplus b, a \oplus b \oplus c]$. Skeletons can also express more complex algorithmic patterns. For example, divide-and-conquer recursion can be expressed as a skeleton $DC(d, e, c)$, whose parameters are a divide function d (mapping a list to two sublists), function e to apply to lists of length one and a conquer function c (to combine two partial results into one).

3 System Implementation

The system sketched in Figure 1 was implemented in Java, using RMI for communication. Java has several advantages for our purposes. First of all, Java bytecodes are portable across a broad range of machines. The skeleton's customizing functional parameters can therefore be used on any of the server machines without rewriting or recompilation. Moreover, Java and RMI provide simple mechanisms for invoking a method (skeleton) remotely on the server.

The interaction between the system components – client, compute server and lookup server – is realized by implementing a set of remote interfaces known to all components. Figure 2 shows a simplified UML class diagram for the most important classes and interfaces of our implementation. Dashed lines with a solid triangle arrowhead connect interfaces and their implementing classes, while open arrowheads denote the "uses" relationship.

Compute Servers provide data-parallel skeletons that are remotely invoked by the clients. For each skeleton, a corresponding interface can be implemented on several servers. For example, for the reduction skeleton, interface `Reduce` is implemented by `ReduceImpl` in Figure 2; for the scan skeleton, interface `Scan` is implemented, etc. Skeleton implementations are adaptable to the server's machine type: e.g. they may be multithreaded Java programs for UMA multiprocessors, MPI programs for clusters, etc. Customizing operators are obtained from the client as classes implementing appropriate remote interfaces, e. g. operators for the reduction skeleton implement the `BinOp` interface. The necessary code shipping is handled transparently by RMI. The system is easily extensible: to add a new skeleton, an appropriate interface must be specified and copied to the codebase, along with any other necessary interfaces (e. g. operators with three parameters). The interfaces can then be implemented on the server and registered with the lookup service in the usual manner.

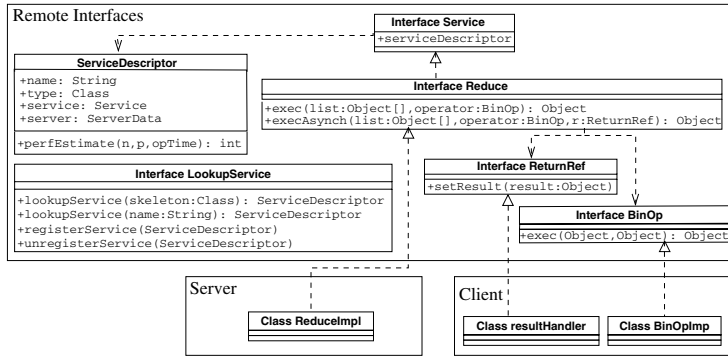


Fig. 2. Simplified class diagram of the implementation

Lookup Service administers the list of available skeletons and is running on its own server. Its hostname and port are known to all servers and clients. Each entry in the list consists of an object of class `ServiceDescriptor`, containing the skeleton's name and the implementing servers, a remote reference to the skeleton's implementation on the server side and a performance-estimation function (cf. Section 4).

Clients and servers interact with the lookup service by calling methods of the `LookupService` interface shown in the class diagram: `registerService` is used by the servers to register their skeletons, and `lookupService` is used by the clients to query for a particular skeleton.

Clients run a simple GUI for program development and server selection. On startup, the lookup service is contacted to obtain a list of all available skeletons. The user can then specify the structure of the program graphically: selected skeletons are displayed as nodes of a directed graph, with an edge between two skeletons if one provides input data for the other (composition). A special skeleton "local" is used to represent local (client-sided) computations in the graph.

From the graphical representation, a partial Java program is generated. It contains all skeleton calls and class definitions for classes that must be implemented by the user (e. g. customizing operators for skeletons). For each pair of server and skeleton, a performance estimate for that pair is computed, as explained in more detail in Section 4. Using this prediction, the user assigns a server to each particular skeleton invocation.

Skeleton Invocation is implemented using Java's RMI mechanism. This has the advantage that all parameter marshalling and unmarshalling as well as code shipping are handled transparently by the RMI system. One drawback, however, is the absence of asynchronous method invocation in RMI. We remedy this situation by providing a second implementation for each skeleton, implementing an asynchronous invocation. The `executeAsynch` method of a skeleton's interface immediately returns a remote reference to an object of class `rObject` which

resides on the server side. To obtain the skeleton's result, the client invokes the `rObject`'s `getResult()` method, which blocks until the results are available. The transmission of data is again handled by RMI.

Skeleton composition is also handled using `rObject`s. For each skeleton, another `execute` method is provided, which receives parameters of type `rObject`. Thus, it is possible to express composition by simply writing
`result=skeleton2.execute(skeleton1.executeAsynch(...));`.
As `executeAsynch` only returns a remote reference, the results are not sent from the server back to the client, and from there on to the next server; instead only remote references are passed on. The second server can then obtain the data directly via the `getResult()` method.

4 Skeleton Performance Prediction

Intuitively, a client should delegate the skeleton execution to a particular Grid server if the skeleton is expected to execute faster on the server than locally or on another server. Invoking a skeleton remotely involves the time costs of sending arguments to and receiving results from the server. Thus, the decision about where to execute a skeleton is influenced by two main factors: performance gain and communication costs.

To decide whether to compute a skeleton remotely – and, if so, then on which server – it is necessary to predict both communication costs and performance gain. Thus, each server in our system provides a function describing the performance of each skeleton implemented by it. A client obtains this function t_{skel} from the lookup service for every server on which the skeleton *skel* is available. The total time T for remote execution can be computed as follows:

$$T = 2t_s + (n + o + r)t_w + t_{skel}(n, p, t_\oplus) \tag{1}$$

n being the size of the parameters, o the size of the operator's bytecode (which must be sent to the server as well) and r the size of the result. The number of processors is p, and $t_\oplus$ is the time taken to execute the customizing operator $\oplus$.

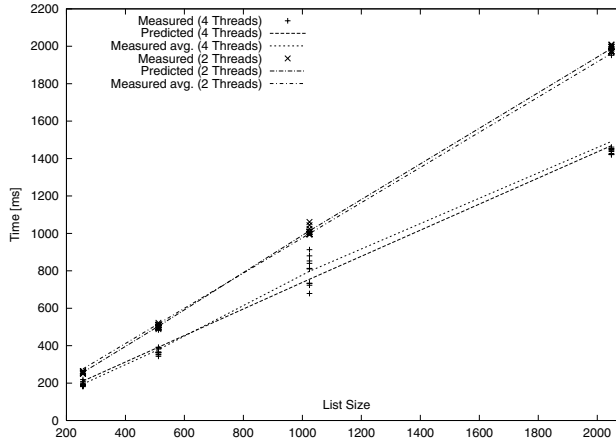
Let us consider the reduce skeleton example. Our experimental parallel implementation of reduction partitions the list into p sublists, with one thread computing the reduction sequentially on each sublist. The partial results are then reduced sequentially in one thread. The time taken to execute the reduce skeleton using this algorithm is

$$t_{red}(n, p, t_\oplus) = (\lceil n/p \rceil - 1)t_\oplus + (p - 1)t_\oplus \tag{2}$$

This algorithm can obviously be improved by organizing the reduction of partial results in a tree-like manner.

We measured the execution time for the reduce skeleton's implementation mentioned above, with 20×20 matrices as list elements and matrix multiplication as the operator. For all measurements, a SUN Ultra 5 Workstation with an UltraSparc-IIi processor running at 360 MHz ("client") and a SunFire

6800 shared-memory SMP system with 16 UltraSparc-III processors at 750 MHz ("server") were used. They are connected via a WAN, the client being at the University of Erlangen and the server at the Technical University of Berlin, with a distance of about 500 km between them.



	t_{red}	t'_{red}	$t_{\oplus}$	t_s	t_w	T_1	T_2
Time [ms]	312.18	320.4	1.21	13.371	0.406	755	797

Fig. 3. Predicted and measured execution time for reduce skeleton

The table in Figure 3 contains the measured time $t_{\oplus}$ for executing the customizing operator (matrix multiplication) on the server. The time t_{red} for reducing a list of 1024 matrices was predicted, using the prediction function for the reduce skeleton given by equation (2) above. The obtained value is quite close to the measured time (t'_{red} in the table), though the latter is slightly higher owing to the overhead for synchronizing threads. The table also contains values t_s and t_w for sending messages via the WAN from Erlangen to Berlin (note that t_w is the time taken to transmit one matrix). The values were measured by sending messages of varying size from the client to the server, without invoking any skeletons. For the total time T for executing the reduce skeleton remotely, two values are given: T_1 is obtained using the predicted value for t_{red} together with equation (1); T_2 is the actual value measured when invoking the reduce skeleton remotely (the average over 10 invocations). The measured value is approximately 5% larger than the predicted one.

In the graph shown in Figure 3, the measured values for lists of sizes 256, 512, 1024 and 2048 are compared to the predicted values. For each list size, the values were measured ten times using the same setting as described above, with two and four threads. The measured values vary considerably (up to 20% for list size 1024 and 4 threads) owing to load changes on the server and varying

network traffic. Thus, the predicted value differs up to 18% from the measured one. Most measured values are, however, much closer to the predicted ones, with differences of only 2 or 3%. Comparing the predicted values to the average values over all ten measurements, the differences are less than 7%.

4.1 Operator Performance Prediction

To achieve realistic time estimates for skeleton execution, it is important to predict accurately both the runtime of the customizing functional arguments (which are Java bytecodes) and the Grid network parameters. While many tools for predicting network performance are available, e. g. the Network Weather Service [6], very little is known about predicting the performance of Java bytecodes.

The simplest way to predict the runtime of an operator would be to send the operator's bytecode to the server, along with a sample set of operands, execute it there and obtain the execution time. Although very accurate time values can be expected, this method consumes a considerable amount of both network and computational resources on the server side, which is a significant drawback. We therefore investigate two approaches that do not involve computations on the server side or communication between client and server: (1) bytecode analysis, and (2) bytecode benchmarking.

Performance Prediction through Bytecode Analysis. To estimate the operator's runtime, we can execute it in a special JVM on the client side, counting how often each instruction is invoked. The obtained numbers for each instruction are then multiplied by a time value for that instruction. Our experiments have shown, that this approach, however, poses the problem of finding valid time values for single instruction runtimes. One way to obtain values that give promising results is reported in [1].

Performance Prediction through Benchmarking. An alternative way of estimating the performance of the operator's Java bytecode is to measure the achievable speedup a priori, by running a benchmark on the server and comparing the result for the benchmark execution on the server side to the runtime of the same benchmark on the client. Then, to obtain an estimate for the execution time of an operator on the server, the time for execution on the client is taken and multiplied by the measured speedup for the benchmark. To increase the accuracy of such performance predictions, several benchmarks containing different instruction mixes (arithmetic instructions, comparison operations, etc.) could be executed on the server. The client then picks the benchmark that appears suitable for the operator in question and uses this to compute a performance estimate as described above.

5 Related Work and Conclusions

Initial research on Grid computing focused, quite naturally, on developing the enabling infrastructure, systems like Globus, Legion and Condor being the promi-

nent examples presented in the seminal "Grid-book" [3]. The next wave of interest focused on particular classes of applications and supporting tools for them, with such important projects as Netsolve [2], GridPP [5] and Cactus.

We feel that algorithmic and programming methodology aspects have been partly neglected at this early stage of Grid research and are therefore not yet properly understood. The main difficulty is the unpredictable nature of the Grid resources, resulting in difficult-to-predict behaviour of algorithms and programs. Initial experience has shown that entirely new approaches to software development and programming are required for the Grid [4].

Our work attempts to overcome the difficulties of algorithm design for Grids by using higher-order, parameterized programming constructs called skeletons. The advantage of skeletons is their high level of abstraction combined with an efficient implementation, tuned to a particular node of the Grid. Even complex applications composed of skeletons have a simple structure, and at the same time each skeleton can exemplify quite a complicated parallel or multithreaded structure in its implementation.

In the experimental programming system described in this paper, we focused on the problem of mapping the parts of an application to particular Grid machines and on the especially challenging problem of performance predictability. The initial results reported here are quite encouraging: they show that even in a heterogeneous, geographically distributed Grid environment the use of skeletons leads to more structured, predictable applications.

The grid system presented in our paper is still in an early stage and lacking many important services, such as resource allocation and authentication services. These issues can be addressed by using a more sophisticated infrastructure, such as provided by the Globus toolkit ([3]).

References

1. M. Alt, H. Bischof, and S. Gorlatch. Program development for computational grids using skeletons and performance prediction. In *Third Int. Workshop on Constructive Methods for Parallel Programming (CMPP 2002)*, Technical Report. Technische Universität Berlin, 2002. To appear.
2. H. Casanova and J. Dongarra. NetSolve: A network-enabled server for solving computational science problems. *Int. J. of Supercomputing Applications and High Performance Computing*, 3(11):212–223, 1997.
3. I. Foster and C. Kesselmann, editors. *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann, 1998.
4. K. Kennedy et al. Toward a framework for preparing and executing adaptive grid programs. In *Proceedings of NSF Next Generation Systems Program Workshop (International Parallel and Distributed Processing Symposium 2002)*, Fort Lauderdale, April 2002.
5. R. Perrot. Testbeds for the GridPP. First US-UK Workshop on Grid Computing.
6. R. Wolski, N. Spring, and J. Hayes. The network weather service: A distributed resource performance forecasting service for metacomputing. *Journal of Future Generation Computing Systems*, 15(5-6):757–768, October 1999.