

Technische Universität Berlin



**Forschungsberichte
der Fakultät IV –
Elektrotechnik und Informatik**

Optimizing the Use of Java RMI for Grid Application Programming

Martin Alt and Sergei Gorlatch

ISSN 1436-9915

No. 2003/08

March 2003

Technische Universität Berlin
Fakultät IV – Elektrotechnik und Informatik
Franklinstraße 28/29 · D-10587 Berlin

Optimizing the Use of Java RMI for Grid Application Programming

Martin Alt and Sergei Gorlatch

Technische Universität Berlin, Sekr. FR5-6,
Franklinstr. 28/29, D-10587 Berlin, Germany
{gorlatch,mnalt}@cs.tu-berlin.de

Abstract. We argue that the traditional RMI (remote method invocation) mechanism causes much unnecessary communication overhead in Grid applications, which run on client machines and outsource time-intensive method calls to high-performance servers. This problem is especially weighty for compositions of remote methods, i. e. where one method uses the result of another method as an argument. We propose three optimizations of RMI at the application level and provide their prototypical implementation on top of Java RMI and an analytical performance model to estimate the achieved overhead reduction. We report experimental results that confirm the performance improvement due to our optimizations on a prototypical Grid system.

1 Introduction

Grid computing aims to combine different kinds of computational resources connected by the Internet and make them easily available to a wide user community. One popular approach to developing applications for Grid-like environments is to provide libraries on high-performance servers, which can be accessed by clients using some remote invocation mechanism, e. g. RPC/RMI [12]. There are several systems that adopt this approach, the libraries' methods being either numerical routines, as in NetSolve [2] and Ninf [8], or generic program components (skeletons), as in our experimental system [1].

An important advantage of keeping libraries of remote methods on servers is that the methods' implementations can be highly optimized for the specific server architecture and hardware. However, such systems also have a serious drawback: to be efficient, they require method calls to be fairly coarse-grained, which should amortize high communication latencies in the wide-area Grid networks. This limits the amount of available concurrency in many important applications, thus reducing the set of potential applications for Grid computing.

In this paper, we argue that the traditional RMI (remote method invocation) mechanism is one of the reasons for the low efficiency of medium- and fine-grained solutions on the Grid. RMI is based on the client/server model of execution, where all parameters for a method are sent from the client to the server and all results are sent back to the client. This leads to much unnecessary communication flow via the client, especially in a typical situation of composing several methods in a program, i. e. when the result of one method call is used as an argument for another method, which is called on the same or another server rather than on the client.

Our goal in this paper is to develop and implement several optimizations of the RMI mechanism at the application level, which would reduce communication overhead and thus improve the efficiency of Grid applications.

The particular contributions and structure of the paper are as follows:

- We present an experimental Grid programming system on top of Java RMI (Section 2).
- We propose three optimizations of the RMI mechanism, together with their prototypical implementation for Java RMI and an analytical performance model (Section 3).
- As an application case study, we discuss the implementation of a solver for linear equation systems using a Java matrix library. We report experimental results for the case study, demonstrating the improvements achieved through our optimizations (Section 4).

We conclude by discussing our results in the context of related work (Section 5).

2 Programming Model on the Grid

We have implemented a Java-based prototypical programming environment for a Grid system consisting of three kinds of components, shown in Fig. 1: clients (left) and servers (right), and a central entity called “lookup service”, used for resource discovery. In this paper, we confine our attention to the programming model and communication between clients and servers; see [1] for details of the system architecture and the issues of resource discovery and management. The particular machines and networks used in our experiments are described in Section 3.6.

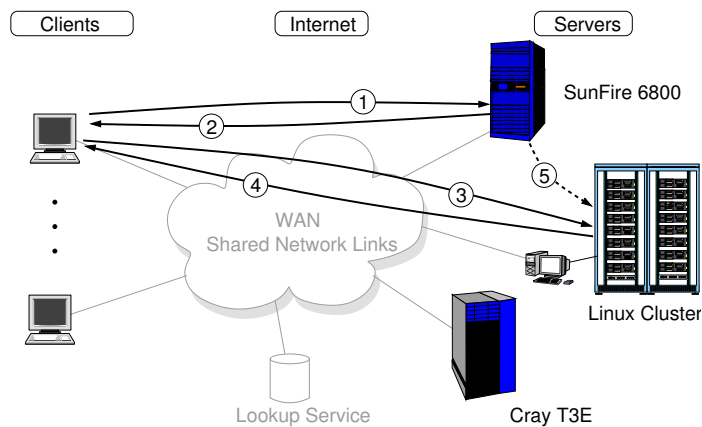


Fig. 1. Experimental Grid system: architecture and interaction

The challenge of Grid programming lies in the fact that the system is highly heterogeneous and dynamic, with servers of different architectures frequently leaving and joining the system. Programs should therefore be potentially executable on as many different servers as possible. Moreover, Grid programs cannot rely on a fixed set of servers: instead, they must be able to adapt to a changing environment and make use of the resources they find available at runtime. In our programming model, each server provides a set of library methods, which are accessible to clients via RMI. Libraries might be specifically developed to be available through RMI, or ex-

isting Java libraries could be “adapted” by writing remote wrapper classes.

For remote execution of a single method, Java’s standard RMI mechanism works well enough: (1) a remote method call is expressed in exactly the same way as a local one, and (2) the server executing the method can be changed at runtime, by changing the corresponding remote reference.

Our work is motivated by many important applications where remote method calls are composed with each other. Throughout this paper, we use a very simple Java code fragment, where the result of `method1` is used as an argument by `method2`:

```
... //get remote reference for server1/2
result1 = server1.method1();
result2 = server2.method2(result1);
```

Fig. 2. Sample Java code: composition of two methods.

The execution of the code shown in Fig. 2 can be distributed: different methods potentially run on different servers, i.e. different RMI references are assigned to `server1` and `server2`. When such a program is executed on the Grid system of Fig 1, methods are called remotely on a corresponding server. If a method’s result is used as a parameter of other remote methods, the result is first sent to the client (arrow ② in the figure), and from there (as a parameter of the second method) to that method’s server (arrow ③). This leads to much unnecessary communication between the client and servers, thus increasing execution time of the program.

It would be more efficient to send the data directly from the first to the second server (⑤ in the figure). The crux of the mechanisms developed in this paper is to optimize distributed composition of methods by enabling such server/server communication, while retaining as far as possible the advantages of programming with RMI mentioned above.

3 Optimizations of RMI and their Implementation

In this section, we begin by presenting the original (plain) RMI and then propose optimizations of RMI for distributed execution of composed method calls. Each of the RMI mechanisms discussed below is presented following the same schema: (1) the idea behind the mechanism, (2) the mechanism’s implementation in Java, and (3) the performance model for the required time.

Our analytical performance model is based on the usual assumption that the time required to send n bytes of data over a network link is as follows: $t_c(n) = t_s + n \cdot t_w$, where t_s is the startup time and t_w the word-transmission time, both determined by the latency and bandwidth of the network used. There is also some overhead for the RMI protocol itself, as well as for serialization and deserialization of objects: we count it in t_s and t_w for constant and variable overhead, respectively.

3.1 Plain RMI

Using plain RMI has the advantage that remote methods are called in exactly the same way as local ones. Thus, the code in Fig. 2 would not change at all when using RMI instead of local methods. The only difference would be that `server1` and `server2` are RMI references, i. e. references to RMI stubs instead of “normal” objects.

However, using plain RMI to execute a composition of methods as in Fig. 2 is not time-efficient because the result of a remote method invocation is always sent back directly to the client. The sequence diagram in Fig. 3 demonstrates that assigning two different servers to `server1` and `server2` in our example code leads to the result of `method1` being sent back to the client, and from there to the second server. Furthermore, even if both methods are executed on the same server, the result is still sent first to the client, and from there back to the server again. For typical applications consisting of many composed RMI calls, this results in very high overhead.

Performance Model In the following, let n_1 and n_2 denote the size of the return values of `method1` and `method2`, respectively. For the

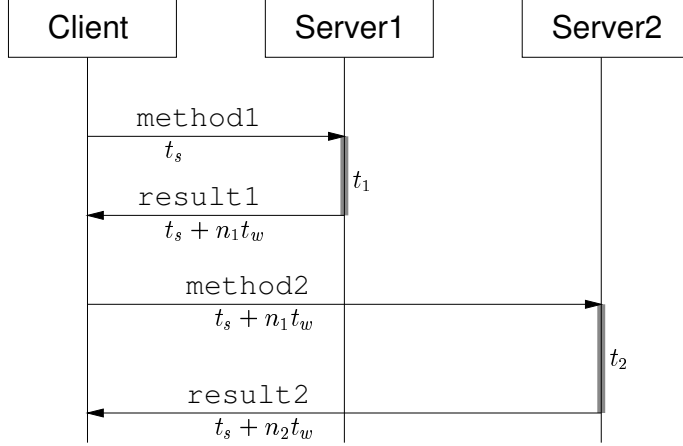


Fig. 3. Distributed composition of methods using plain RMI

sake of simplicity, we assume that parameters t_s and t_w are the same for all involved network links, so clients and servers are assumed to be at “equal distance” from each other (corresponding generalization is straightforward).

The communication time for our sample program with two remote method invocations consists of two parts: for `method1`, the result (n_1 bytes) is sent to the client; for `method2`, this first result is sent as a parameter to the server, and the second result is sent back to the client ($n_1 + n_2$ bytes). Using the notation presented above, overall communication time is expressed by

$$T_{plain} = 2t_s + n_1 t_w + 2t_s + (n_1 + n_2)t_w = 4t_s + (2n_1 + n_2)t_w \quad (1)$$

3.2 Optimization 1 : Lazy RMI

Our first optimization, called *lazy RMI*, aims to reduce the amount of data sent from the server to the client upon method completion. Instead of the result being sent to the client, a remote reference is returned. The client passes this reference on to the next server, which uses the reference to request the result. This is depicted in Fig. 4, with dotted horizontal lines for sending references.

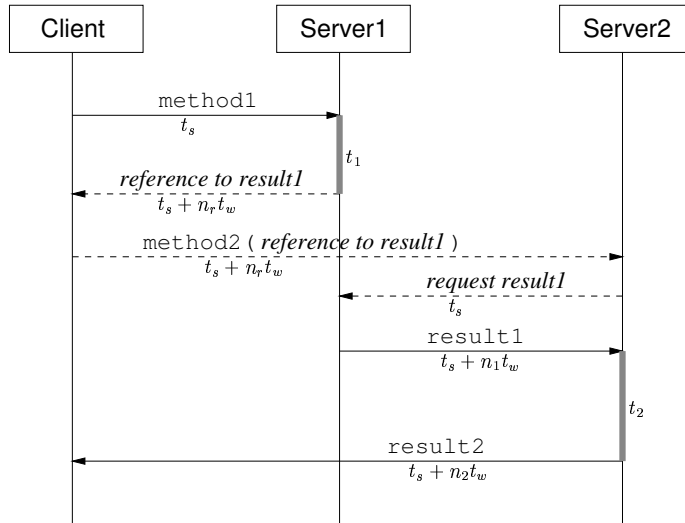


Fig. 4. Distributed composition using lazy RMI with remote references

Implementation Under lazy RMI, methods do not return the result of the computations directly. Instead, they return a remote reference to an object of the new class `RemoteReference`. This class is instantiated on the server side and provides two methods:

```
public Object getValue() ...; //remote method
public void setValue(Object o) ...;
```

The `setValue()` method is called by the application method's implementation when the result of the method is available, passing the result as a parameter. Method `getValue()` is used by the next application method (or by the client) to retrieve this result and may be called remotely. Calls to `getValue()` block until a value has been assigned using `setValue()`. If `getValue()` is called remotely via RMI, the result is sent over the network to the next server.

On the server side, all parameters of type `RemoteReference` are unpacked using the `getValue()` method of the passed reference and the results are wrapped into a new `RemoteReference`. Thus, the code for `method2` from Fig. 2 is as follows, using `RemoteReference`:

```
method2(RemoteReference ref) {
    Object result1 = ref.getValue();
```

```

// do computations using result1
return result2;}

```

From the user’s point of view, a distributed composition of methods under lazy RMI is expressed in the same way as with plain RMI.

Performance Model Let n_r denote the size of a remote reference passed over the network. Then, the communication time of the remote composition under lazy RMI can be expressed as

$$\begin{aligned}
T_{distributed} &= 2t_s + n_r t_w + 2t_s + (n_r + n_2)t_w + 2t_s + n_1 t_w \\
&= 6t_s + (2n_r + n_1 + n_2)t_w
\end{aligned} \tag{2}$$

in the distributed case (each method executed on a different server). The use of remote references reduces the transmitted data from $2n_1 + n_2$ to $2n_r + n_1 + n_2$ bytes, at the expense of an additional RMI call (to transmit `result1` between the servers). If $n_1 > 2n_r$, then the runtime is improved.

If both methods in the example code are executed on the same server, the result of the first method is already available locally. Therefore, when `method2` requests the result of `method1`, it is not sent via the network because it can be copied locally on the server. However, `method2` still accesses the result of `method1` by calling the `getValue` method on the *remote reference* using RMI, which incurs overhead for serializing and deserializing the data. Using parameter $\tau_{s/w}$ for startup and “transmission” time for serialization and sending through a local socket, the communication costs amount to

$$\begin{aligned}
T_{remote} &= 2t_s + n_r t_w + 2t_s + (n_r + n_2)t_w + 2\tau_s + n_1 \tau_w \\
&= 4t_s + (2n_r + n_2)t_w + 2\tau_s + n_1 \tau_w
\end{aligned} \tag{3}$$

Compared to plain RMI, the amount of data sent over the network changes from $2n_1 + n_2$ to $2n_r + n_2$, thus reducing communication cost if $2n_r < n_1$. The time for sending over a local socket can be assumed to be smaller than the time for sending over a network, as the data is copied between two memory locations. So, $\tau_{s/w} < t_{s/w}$, further decreasing communication time.

3.3 Optimization 2 : Localized RMI

Despite using remote references, the runtime when both methods in Fig. 2 are executed on one server is still quite large because all

communications occur through local sockets. As both endpoints of the connection are on the same host, there is no network access involved, but there is still substantial overhead for serializing and deserializing the data and sending it through the socket.

To avoid this overhead, we propose our next optimization, called *localized RMI*. The idea is to check, for every access to a remote reference, whether the data is available locally or not. In the local case, the object is returned directly without issuing an RMI call, thus reducing the runtime.

Implementation To prevent unnecessary transmissions of data over local sockets, the implementation of the `getValue()` method checks if the requested object is available locally and, if that is the case, returns a local reference. To achieve this, class `RemoteReference` contains the IP address of the object’s server and the (standard Java) hashvalue of the object, thus uniquely identifying it. When `getValue()` is invoked, it first checks if the IP address is the address of the local server where `getValue()` is invoked. If so, it uses the hashvalue as a key for a local hashtable (which is static for class `RemoteReference`) to obtain a local reference to the object. This reference is then returned to the calling method.

Note that `RemoteReference` is not allowed to be a remote (RMI) class, because `getValue()` must be executed locally rather than via RMI. We introduce the class `RemoteValue` (having the same methods as `RemoteReference`), accessible remotely. Each instance of `RemoteReference` has a reference to a `RemoteValue` instance, which is used to retrieve an object from a remote host if it is not available locally. The translation of remote to local references is handled automatically by the `RemoteReference` implementation, so the application program remains the same as in Section 3.2.

Performance Model The communication costs can be estimated as follows:

$$\begin{aligned}\hat{T}_{remote} &= 2t_s + n_r t_w + 2t_s + (n_r + n_2)t_w + 2\tau_s + n_1\tau_w \\ &= 4t_s + (2n_r + n_2)t_w\end{aligned}\quad (4)$$

Thus, compared to plain RMI, the amount of transmitted data is reduced from $2n_1 + n_2$ to $2n_r + n_2$.

3.4 Optimization 3 : Future-based RMI

Returning to our example code in Fig. 2, even if both methods are executed on the same server, the second method cannot be executed until the remote reference for the result of the first method call has been sent to the client and back. Thus, the execution of the second method is delayed, although the data is already available on the server.

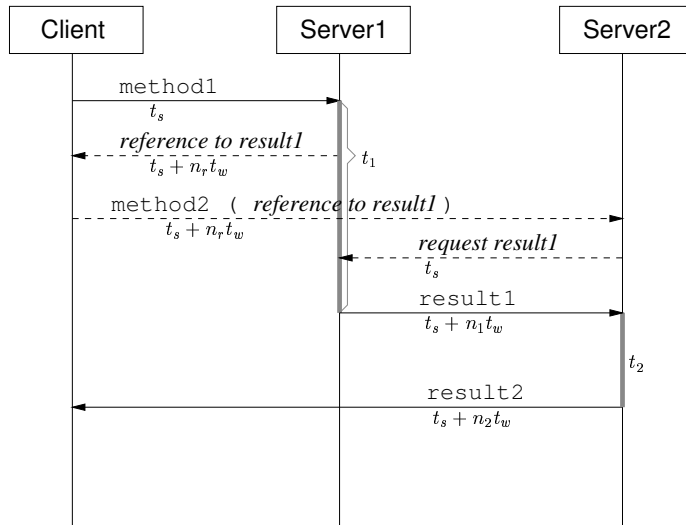


Fig. 5. Distributed composition using remote references and future-based RMI

We see this unnecessary delay as an additional source of optimization, which we call *future-based RMI*. The idea is that all method invocations immediately return a remote reference to the result, a so-called future reference, which is sent to the client and can be passed on to the next method (an attempt to retrieve thus referenced data is blocked until the data becomes available). As a result, computations and communication between client and server overlap (Fig. 5).

Implementation To execute a method after returning a future reference to the caller, remote methods on the server spawn a new thread to carry out computations and then return immediately. This

is shown as pseudocode for `method1` in Figure 6. To avoid thread-creation overhead, a pool of threads is created when the server starts; threads are taken from the pool on demand.

```
public RemoteReference method1(...) {
    RemoteReference ret = new RemoteReference();
    /* execute method1 in a new thread, that will
       call ret.setValue() upon completion */
    Thread t = new Method1Thread(..., ret);
    return ret;}

```

Fig. 6. Pseudocode for asynchronous `method1` returning a future reference

On the client side, there are no changes to the application program when using future-based RMI.

Performance Model: If t_i denotes the runtime for `method i` , the overall runtime for the sample program using future-based RMI, and thus asynchronous method invocation, is

$$T_{asynch} = t_s + \max(t_1, 2(t_s + n_r t_w) + t_s) + n_1 t_w + t_2 + t_s + n_2 t_w \quad (5)$$

Assuming $t_1 > 2(t_s + n_r t_w) + t_s$ (otherwise, the cost for communicating between client and server would be higher than the computation itself, making remote execution pointless), this results in

$$T_{asynch} = t_s + t_1 + n_1 t_w + t_2 + t_s + n_2 t_w \quad (6)$$

which is equivalent to the time that would be necessary for calling the second method via RMI from the first server, instead of calling it from the client.

In the case of both methods running on the same server, the runtime is further reduced to

$$\hat{T}_{asynchL} = t_s + t_1 + t_2 + t_s + n_2 t_w \quad (7)$$

which is equivalent to executing a single RMI method with runtime $t_1 + t_2$ on the server instead of two methods. Thus, the overhead for the second RMI call becomes completely hidden.

The overhead for starting a method in its own thread is not considered in our model. However, this overhead is very small (less than $75\ \mu\text{s}$ in our implementation) and can therefore be neglected.

3.5 When to use the optimizations

The three optimizations presented above are not orthogonal to each other. The localized RMI is an extension of lazy RMI: if methods are distributed over different servers, lazy and localized RMI are equivalent, while for methods running on the same server, localized RMI always outperforms the unoptimized lazy version. Thus, there is no reason to use lazy RMI without the localized optimization. Using future-based RMI means using lazy RMI too, because the remote references used with the lazy version are necessary to retrieve the result of the method upon completion. On the other hand, future-based RMI hides communication times, so using lazy RMI with asynchronous method calls instead of synchronous calls always reduces total completion time.

All three optimizations should therefore be used together (we call the result of combined optimizations *improved RMI*). We discuss, whether to use plain or improved RMI for two cases:

Remote composition on one server: If both methods are executed on the same server, and the result of the first method is larger than a remote reference (i. e. $n_1 > n_r$), then remote references should be used. Since remote references are usually small (32 bytes in our prototypical implementation), a rule of thumb is to use improved RMI whenever a method returns an object instead of a primitive data type. The only exception is made for objects that have only one member variable, such as `java.lang.Integer`.

Distributed composition over several servers: Improved RMI should be used if the method's runtime exceeds the time needed for sending a remote reference from the server to the client and on to the next server, i. e. $t_1 > 2(t_s + n_r t_w)$. The right-hand side of the inequality is relatively small (less than 100 ms in our system), so most remote methods can be expected to take longer. If communication takes more time than computation on the server,

it is better to execute the method locally instead of sending it to a server anyway.

Thus, in most cases it is advantageous to replace plain RMI with improved RMI.

3.6 Experimental Results

Our testbed environment has the structure shown in Figure 1 and consists of two university LANs, one at TU Berlin and the other at the University of Passau. They are connected by the German Academic Internet Backbone (WiN), covering a distance of approx. 700 km. We used the multiprocessor SunFire 6800 located in Berlin as our server and a Pentium 200 MHz as the client in Passau, both using SUN's JDK1.4.1 (HotSpot Client VM). For the network link, we measured a startup time of $t_s = 21$ ms and a word-transmission time of $t_w = 323$ μ s using byte-arrays.

We measured the performance of the small sample program from Figure 2, with `method1` and `method2` both taking 500 ms, and the amount of data sent over the network ranging between 10 KB and 100 KB (stored in byte arrays of the appropriate size). Figure 7 shows the runtimes for three different versions of the program: (1) two method calls with plain RMI, (2) two method calls with improved RMI, and (3) one method call which takes twice as much time as the original method call. We regard the one-method version as providing ideal runtime ("lower bound") for a composition of remote methods. The figure presents five measurements for each version of the program, with the average runtimes for each parameter size connected by lines for the clarity sake.

The figure shows that the improved RMI version is between 100 ms and 350 ms faster than the plain RMI. Considering only communication times (i. e. subtracting one second for computations on the server side), the time for plain RMI is approximately twice as long as for the improved version. This means that the communication time for the second, composed method call is almost completely hidden owing to the laziness and asynchrony introduced by our optimizations. The composition under improved RMI is only 10-15 ms slower than the "lower-bound" version, which means that our optimizations eliminated between 85% and 97% of the original overhead.

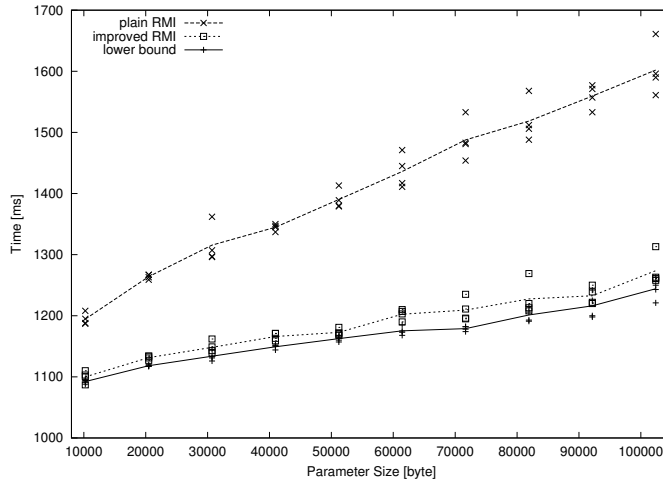


Fig. 7. Runtimes for the example in Fig. 2 using plain and improved RMI

3.7 Exception handling

With plain RMI, exception handling resembles local method invocation: if an exception is thrown in the remote method, it is caught on the server, then sent to the client and rethrown there.

If our optimized RMI is used, a remote method invocation returns to the calling client before the method on the server is actually completed; thus, an exception thrown on the server cannot be thrown immediately on the client. In our implementation, the exception is caught on the server and stored in the object, that the future reference returned to the client points to. When the `getValue()` method of this object is called, the exception is rethrown and propagated to the caller (either the client or another server). If the caller of `getValue()` is another server-sided method, the exception is caught again and wrapped in the next object referenced by a future reference until it finally reaches the client.

4 Application Case Study

We have tested our approach on a linear system solver, implemented using the matrix library Jama ([3]). For a given system $Ax = b$, we look for vector \hat{x} that minimizes $\chi^2 = (Ax - b)^2$. Our example

uses the singular value decomposition method (SVD): the SVD of a matrix is a decomposition $A = U \cdot \Sigma \cdot V^T$, U and V being orthonormal matrices and Σ a diagonal matrix (see e.g. [10]). The inverse A^{-1} can be computed as $A^{-1} = V \cdot \Sigma^+ \cdot U^T$. The matrix Σ^+ is obtained from matrix Σ by replacing each value σ_i on the diagonal with its reciprocal value $\sigma_i^+ = 1/\sigma_i$. If matrix A is singular, then some values on the diagonal of Σ are zero. For these values, the corresponding entry in Σ^+ is set to $\sigma_i^+ = 0$. The solution \hat{x} is then computed by evaluating $A^{-1} \cdot b$.

The solver works in four steps: first the SVD for the input matrix A is computed, using a Jama library method. Then the inverse of A is computed by transforming Σ to Σ^+ using a user supplied method and computing $V \cdot \Sigma^+ \cdot U^T$ using Jama methods. In the next step, A^{-1} is multiplied by b to obtain x , and finally, the residual $r = |A \cdot x - b|$ is computed.

```

Matrix A,x,b,U,S,V,err;
SVD svd; double r;
//decompose
svd = srv.svd(A);
U = srv.s_getU(svd); S = srv.s_getS(svd);
V = srv.s_getV(svd);
//adjust
S=adjust(S);
//compute result
x = srv.times(V, srv.times(S,
                        srv.times(srv.transpose(U), b)));
err = srv.minus(srv.times(A,x),b);
r = srv.normInf(err);

```

Fig. 8. Case study using the Jama library and plain Java RMI

The code for the application using plain RMI is shown in Fig. 8. The `srv` variable holds the RMI reference to a remote object on the server, providing access to the Jama methods. As the Jama library does not provide methods for transforming Σ to Σ^+ , this is done using a local method `adjust`, which is executed on the client side. All matrix operations are carried out by remotely calling methods

on the server side. All methods, with exception of `srv.svd` at the beginning, have parameters that are results of other methods. Thus, the program makes substantial use of method composition.

```

RemoteReference rA,rx,rb,U,S,V,err,svd,rr;
Matrix A,b,x; SVD svd; double r;
rA=new RemoteReference(A);
rb=new RemoteReference(b);
//decompose
svd = srv.svd(rA);
U = srv.s_getU(svd); S = srv.s_getS(svd);
V = srv.s_getV(svd);
S=srv.execTask(new Adjust(),S); //adjust
//compute result
rx = srv.times(V, srv.times(S,
    srv.times(srv.transpose(U), rb)));
err = srv.minus(srv.times(rA,x),rb);
rr = srv.normInf(err);
x = rx.getValue();
r = rr.getValue();

```

Fig. 9. Case study using the Jama library and *improved RMI*

The version of the program using the improved RMI is shown in Figure 9. There are two changes compared with the plain RMI version. First, the parameters A and b are packed into `RemoteReferences` for use with the server-sided Jama methods, the results x and r are returned as `RemoteReferences` (`rr` and `rx`), and the `getValue()` method needs to be called to obtain the results. Second, the `adjust` method for transforming Σ to Σ^+ is sent for execution on the server. This is achieved by encapsulating the method in its own object (of type `Adjust`), which is executed on the server by providing it as a parameter to the server-sided method `execTask`. This improves performance because it would take more time to send matrix S over the network and execute it on the client than to send the code to the server and execute it there. With plain RMI, performance would decrease if the method were sent to the server.

The performance of the program was measured using the same hardware platform and JDK as in Section 3.6. Fig. 10 shows the runtimes of three versions of the solver: plain RMI, improved RMI, and the version running completely on the server side (“ideal”).

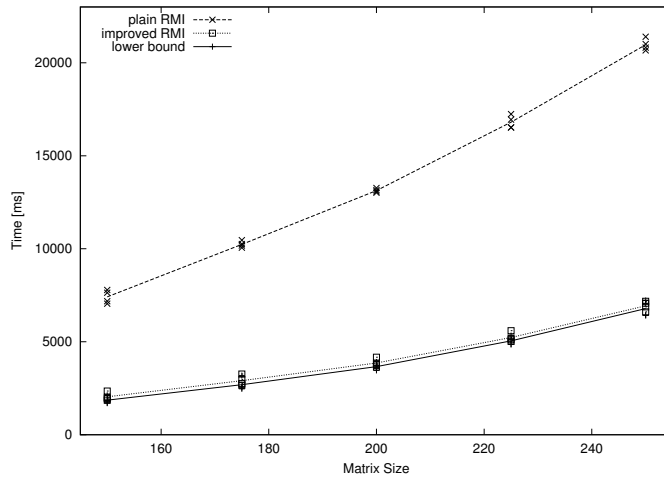


Fig. 10. Measured runtimes for the case study

The measurements for the case study confirm the results for the simple example program presented in Section 3.6. Owing to fine-grained remote methods in the case study, the plain RMI version for their composition is much (three to four times) slower than the “ideal” version, which runs completely on the server side. By contrast, the improved RMI version is able to eliminate most of the overhead and is only less than 10% slower than the ideal version. Thus, using improved RMI, a composition of comparatively fine-grained methods can be executed efficiently.

5 Conclusions and Related Work

We have presented three optimizations of the remote method invocation (RMI) mechanism in the context of Grid computing. The proposed solutions facilitate more efficient remote execution of compositions of computation-intensive methods on high-performance servers

and broaden the class of applications efficiently implementable on the Grid. This extends the scope of Grid systems compared with approaches like NetSolve [2] or Ninf [8], which use RPC and are efficient only for coarse-grained applications. Our prototypical implementation confirmed the advantages of the proposed optimizations in the case of Sun's Java RMI, and can thus be viewed as proof of the concept's viability.

Several research efforts have been made to facilitate the use of RMI in the context of parallel and distributed programming. Most of these aim at improving performance by re-implementing RMI and/or the serialization protocol. Prominent examples are Manta [7] using native code implementations, KaRMI [9] introducing a more efficient runtime system written in Java, and Ibis [13], which uses a special compiler to generate serialization code at compile time. Other approaches ([5], [4]) improve performance by replicating or caching objects on the client side and executing methods locally on the client or using asynchronous method calls [11]. A different approach is taken by [6], where RMI is augmented with collective communication.

An important advantage of our approach is that it is orthogonal to the underlying RMI implementation. Our optimizations are thus directly applicable to these faster RMI systems, too, and can be used along with them without any changes being made.

The novelty of our work is that, whereas previous research dealt with single or repeated RMI calls, we focus on an efficient execution of composed method calls, where the result of one call is an argument of another. This situation is very typical of many Grid applications, and our work has demonstrated several opportunities to improve the performance of such calls.

One drawback of the implementation presented in this paper is that static type checking is limited to local methods, because all parameters and result values of the "improved RMI" are of type `RemoteReference`. To get and set a `RemoteReferences` value, the `getValue` and `setValue` methods are used, requiring values to be cast to type `java.lang.Object` by hand. Thus, the compiler cannot perform any static type checking. This problem can be eliminated by creating a `RemoteReference` class for all classes used, in much the same way that Java RMI uses `rmic` to create stub classes for classes accessed remotely.

Another disadvantage of the user-level implementation is that “dereferencing” of a remote reference is done explicitly by the application programmer calling the `getValue` method of `RemoteReference`. We plan to remedy this by implementing a runtime system that automatically retrieves the remote object on the first access to the remote reference.

Acknowledgements We wish to thank Thilo Kielmann and Robert V. van Nieuwpoort for their very helpful comments on the preliminary version of this paper.

References

- [1] M. Alt, H. Bischof, and S. Gorlatch. Algorithm design and performance prediction in a Java-based Grid system with skeletons. In B. Monien and R. Feldmann, editors, *Euro-Par 2002*, volume 2400 of *Lecture Notes in Computer Science*, pages 899–906. Springer, August 2002.
- [2] D. Arnold, S. Agrawal, S. Blackford, J. Dongarra, M. Miller, K. Seymour, K. Sagi, Z. Shi, and S. Vadhiyar. Users’ guide to netsolve v1.4.1. Innovative Computing Dept. Technical Report ICL-UT-02-05, University of Tennessee, Knoxville, TN, June 2002.
- [3] J. Hicklin, C. Moler, P. Webb, R. F. Boisvert, B. Miller, R. Pozo, and K. Remington. JAMA: A Java matrix package. <http://math.nist.gov/javanumerics/jama/>.
- [4] V. Krishnaswamy, D. Walther, S. Bhola, E. Bommaiah, G. Riley, B. Topol, and M. Ahamad. Efficient implementations of Java remote method invocation (RMI). In *Usenix Conference on Object-Oriented Technologies and Systems (COOTS)*, pages 19–36, April 1998.
- [5] J. Maassen, T. Kielmann, and H. Bal. Efficient replicated method invocation in Java. In *Java Grande Conference*, pages 88–96. ACM, 2000.
- [6] J. Maassen, T. Kielmann, and H. E. Bal. GMI: Flexible and efficient group method invocation for parallel programming. In *LCR '02: Sixth Workshop on Languages, Compilers, and Runtime Systems for Scalable Computers*, 2002.

- [7] J. Maassen, R. van Nieuwpoort, R. Veldema, H. Bal, T. Kielmann, C. Jacobs, and R. Hofman. Efficient Java RMI for parallel programming. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 23(6):747 – 775, November 2001.
- [8] H. Nakada, M. Sato, and S. Sekiguchi. Design and implementations of Ninf: towards a global computing infrastructure. *FGCS*, 15(5-6):649–658, 1999.
- [9] M. Philippsen, B. Haumacher, and C. Nester. More efficient serialization and RMI for Java. *Concurrency: Practice and Experience*, 12(7):495–518, May 2000.
- [10] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery. *Numerical Recipes in C: The Art of Scientific Computing*. Cambridge University Press, second edition, 1992.
- [11] R. Raje, J. Williams, and M. Boyles. An asynchronous remote method invocation (ARMI) mechanism in Java. *Concurrency: Practice and Experience*, 9(11):1207–1211, 1997.
- [12] K. Seymour, H. Nakada, S. Matsuoka, J. Dongarra, C. Lee, and H. Casanova. Overview of GridRPC: A remote procedure call API for Grid computing. In M. Parashar, editor, *Grid Computing - GRID 2002, Third International Workshop*, volume 2536 of *LNCS*, pages 274–27. Springer, November 2002.
- [13] R. van Nieuwpoort, J. Maassen, R. Hofman, T. Kielmann, and H. Bal. Ibis: an efficient Java-based Grid programming environment. In *JavaGrande ISCOPE*. ACM, November 2002.