Wide-Area Parallel Programming using the Remote Method Invocation Model

Rob van Nieuwpoort Jason Maassen Henri E. Bal
Thilo Kielmann Ronald Veldema

Department of Computer Science, Vrije Universiteit, Amsterdam, The Netherlands

rob@cs.vu.nl jason@cs.vu.nl bal@cs.vu.nl kielmann@cs.vu.nl rveldema@cs.vu.nl

http://www.cs.vu.nl/albatross/

Abstract

Java's support for parallel and distributed processing makes the language attractive for metacomputing applications, such as parallel applications that run on geographically distributed (wide-area) systems. To obtain actual experience with a Java-centric approach to metacomputing, we have built and used a high-performance wide-area Java system, called Manta. Manta implements the Java Remote Method Invocation (RMI) model using different communication protocols (active messages and TCP/IP) for different networks. The paper shows how wide-area parallel applications can be expressed and optimized using Java RMI. Also, it presents performance results of several applications on a wide-area system consisting of four Myrinet-based clusters connected by ATM WANs. We finally discuss alternative programming models, namely object replication, JavaSpaces, and MPI for Java.

1 Introduction

Metacomputing is an interesting research area that tries to integrate geographically distributed computing resources into a single powerful system. Many applications can benefit from such an integration [14, 32]. Metacomputing systems support such applications by addressing issues like resource allocation, fault tolerance, security, and heterogeneity. Most metacomputing systems are language-neutral and support a variety of programming languages. Recently, interest has also arisen in metacomputing architectures that are centered around a single language. This approach admittedly is restrictive for some applications, but also has many advantages, such as a simpler design and the usage of a single type system. In [36], the advantages of a Java-centric approach to metacomputing are described, including support for code mobility, distributed polymorphism, distributed garbage collection, and security.

In this paper, we describe our early experiences in building and using a high-performance Java-based system for one important class of metacomputing applications: parallel computing on geographically distributed resources. Although our system is not a complete metacomputing environment yet (e.g. it currently provides neither code mobility nor fault tolerance), it is interesting for several reasons. The system, called *Manta*, focuses on optimizations to achieve high performance with Java. It uses a native compiler and an efficient, light-weight RMI (Remote Method Invocation) protocol that achieves a performance close to that of C-based RPC protocols [25]. We have implemented Manta on a geographically distributed system, called *DAS*, consisting of four Pentium Pro/Myrinet cluster computers connected by wide-area ATM links. The resulting system is an interesting platform for studying parallel Java applications on geographically distributed systems.

The Java-centric approach achieves a high degree of transparency and hides many details of the underlying system (e.g., different communication substrates) from the programmer. For several high-performance applications, however, the huge difference in communication speeds between the local and wide-area networks is a problem. In our DAS system, for example, a Java RMI over the Myrinet LAN costs about 40 μsec, while an RMI over the ATM WAN costs several milliseconds. Our Java system therefore exposes the structure of the wide-area system to the application, so applications can be optimized to reduce communication over the wide-area links.

This paper is based on our earlier work as published in [34]. We show how wide-area parallel applications can be expressed and optimized using Java RMI and we discuss the performance of several parallel Java applications on DAS. We also discuss some shortcomings of the Java RMI model for wide-area parallel computing and how this may be overcome by adapting features from alternative programming models. The outline of the paper is as follows. In Section 2 we describe the implementation of Manta on our wide-area system. In Section 3 we describe our experiences in implementing four wide-area parallel applications in Java and we discuss their performance. In Section 4 we discuss which alternative programming models may contribute to an RMI-based programming model. In Section 5 we look at related work and in Section 6 we give our conclusions.

2 A wide-area parallel Java system

In this section, we will briefly describe the DAS system and the original Manta system (as designed for a single parallel machine). Next, we discuss how we implemented Manta on the wide-area DAS system. Finally, we compare the performance of Manta and the Sun JDK on the DAS system.

2.1 The wide-area DAS system

We believe that high-performance metacomputing applications will typically run on collections of parallel machines (clusters or MPPs), rather than on workstations at random geographic locations. Hence, metacomputing systems that are used for parallel processing will be *hierarchically* structured. The DAS experimentation system we have built reflects this basic assumption, as shown in Figure 1. It consists of four clusters, located at different universities in The Netherlands. The nodes within the same cluster are connected by 1.2 Gbit/sec Myrinet [7]. The clusters are connected by dedicated 6 Mbit/s wide-area ATM networks.

The nodes in each cluster are 200 MHz/128 MByte Pentium Pros. One of the clusters has 128 processors, the other clusters have 24 nodes each. The machines run RedHat Linux version 2.0.36. The Myrinet network is a 2D torus and the wide area ATM network is fully connected. The system, called DAS, is more fully described in [29] (and on *http://www.cs.vu.nl/das/*).

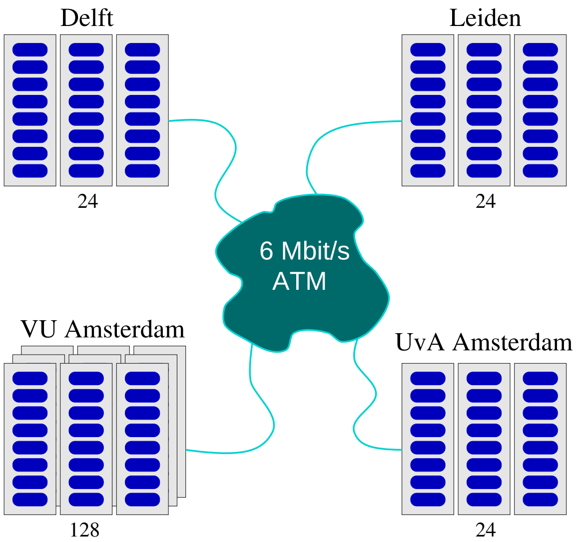

Figure 1: The wide-area DAS system.

2.2 The Manta system

Manta is a Java system designed for high-performance parallel computing. Like JavaParty [28], Manta uses a separate *remote* keyword to indicate which classes allow their methods to be invoked remotely. This method is somewhat more flexible and easier to use than inheriting from *java.rmi.server.UnicastRemoteObject* (the standard RMI mechanism). JavaParty requires a preprocessor for implementing this language extension; for Manta, we have modified our compiler. Except for this difference, the programming model of Manta is the same as that of standard RMI. Manta uses a native compiler and an optimized RMI protocol. The most important advantage of a native compiler (compared to a JIT) is that it can do more aggressive optimizations and therefore generate better code. The compiler also generates the serialization and deserialization routines, which greatly reduces the runtime overhead of RMIs. Manta nodes thus contain the executable code for the application and (de)serialization routines. The nodes communicate with each other using Manta's own light-weight RMI protocol.

The most difficult problem addressed by the Manta system is to allow interoperability with other JVMs. One problem is that Manta has its own, light-weight RMI protocol that is incompatible with Sun's JDK protocol. We solve this problem by letting a Manta node also communicate through a JDK-compliant protocol. Two Manta nodes thus communicate using our fast protocol, while Manta-to-JVM RMIs use the standard RMI protocol.

Another problem concerning interoperability is that Manta uses a native compiler instead of a byte code interpreter (or JIT). Since Java RMIs are polymorphic [35], Manta nodes must be able to send and receive byte codes to interoperate with JVMs. For example, if a remote method expects a parameter of a certain class *C*, the invoker may send it an object of a subclass of *C*. This subclass may not yet be available at the receiving Manta node, so its byte code may have to be retrieved and integrated into the computation. With Manta, however, the computation is an executable program, not a JVM. In the reverse situation, if Manta does a remote invocation to a node running a JVM, it must be able to send the byte codes for subclasses that the receiving JVM does not yet have.

Manta solves this problem as follows. If a remote JVM node sends byte code to a Manta node, the byte code is compiled dynamically to object code and this object code is linked into the running application using the `dlopen()` dynamic linking interface. Also, Manta generates byte codes for the classes it compiles (in addition to executable code). These byte codes are stored at an http daemon, where remote JVM nodes can retrieve them. For more details, we refer to [25].

The Manta RMI protocol is designed to minimize serialization and dispatch overhead, such as copying, buffer management, fragmentation, thread switching, and indirect method calls. Manta avoids the several stream layers used for serialization by the JDK. Instead, RMI parameters are serialized directly into a communication buffer. Moreover, the JDK stream layers are written in Java and their overhead thus depends on the quality of the interpreter or JIT. In Manta, all layers are either implemented as compiled C code or compiler-generated native code. Heterogeneity between little-endian and big-endian machines is achieved by sending data in the native byte order of the sender, and having the receiver do the conversion, if necessary.[1] For further optimization, the Manta compiler heuristically checks whether a method called via RMI may block during execution. If the compiler can exclude this, remote invocations are served without thread creation at the server side. Otherwise, threads from a thread pool are used to serve remote method invocations.

To implement distributed garbage collection, the Manta RMI protocol also keeps track of object references that cross machine boundaries. Manta uses a mark-and-sweep algorithm (executed by a separate thread) for local garbage collection and a reference counting mechanism for remote objects.

The serialization of method arguments is an important source of overhead of existing RMI implementations. Serialization takes Java objects and converts (serializes) them into an array of bytes. The JDK serialization protocol is written in Java and uses reflection to determine the type of each object during run time. With Manta, all serialization code is generated by the compiler, avoiding the overhead of dynamic type inspection. The compiler generates a specialized serialization and deserialization routine for every class. Pointers to these routines are stored in the method table. The Manta serialization protocol optimizes simple cases. For example, an array whose elements are of a primitive type is serialized by doing a direct memory-copy into the message buffer, which saves traversing the array. Compiler generation of serialization is one of the major improvements of Manta over the JDK [25].

In the current implementation, the classes of serialized objects are transferred as numerical type identifiers. They are only consistent for the application binary for which they have been generated by the Manta compiler. This implies that Manta's RMI is type safe as long as all nodes of a parallel run execute the same binary. When combining different application binaries, type inconsistencies may occur. Due to the same problem, Manta's serialization may currently only be used for persistent storage when the same application binary stores and later loads serialized objects. The replacement of Manta's numerical type identifiers by a globally unique (and hence type safe) class identification scheme is subject to ongoing work.

[1] Manta supports the serialization and deserialization protocols needed to support heterogeneous systems, but the underlying Panda library does not yet support heterogeneity, as it does not do byte-conversions on its headers yet.

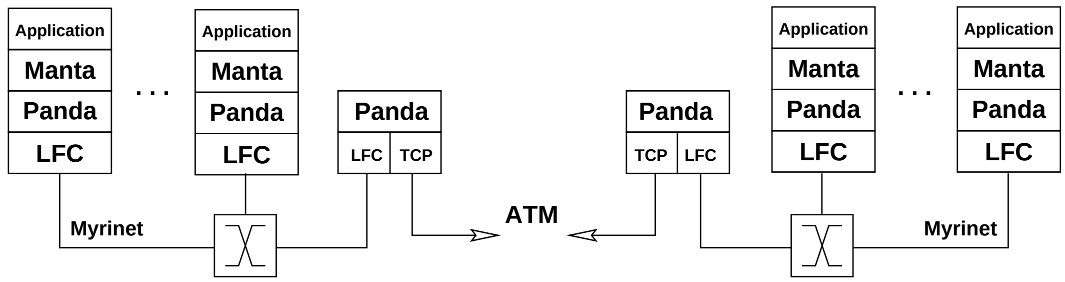

Figure 2: Wide area communication based on Panda

2.3 Manta on the wide area DAS system

To implement Java on a wide-area system like DAS, the most important problem is how to deal with the different communication networks that exist within and between clusters. As described in Section 2.1, we assume that wide-area parallel systems are hierarchically structured and consist of multiple parallel machines (clusters) connected by wide area networks. The LANs (or MPP interconnects) used within a cluster typically are very fast, so it is important that the communication protocols used for intra-cluster communication are as efficient as possible. Inter-cluster communication (over the WAN) necessarily is slower.

Most Java RMI implementations are built on top of TCP/IP. Using a standard communication protocol eases the implementation of RMI, but also has a major performance penalty. TCP/IP was not designed for parallel processing, and therefore has a very high overhead on fast LANs such as Myrinet. For the Manta system, we therefore use different protocols for intra-cluster and inter-cluster communication.

To obtain a modular and portable system, Manta is implemented on top of a separate communication library, called Panda [3]. Panda provides communication and multithreading primitives that are designed to be used for implementing runtime systems of various parallel languages. Panda's communication primitives include point-to-point message passing, RPC, and broadcast. The primitives are independent of the operating system or network, which eases porting of languages implemented on top of Panda. The implementation of Panda, however, is structured in such a way that it can exploit any useful functionality provided by the underlying system (e.g., reliable message passing or broadcast), which makes communication efficient [3].

Panda has been implemented on a variety of machines, operating systems, and networks. The implementation of Manta and Panda on the wide-area DAS system is shown in Figure 2. For intra-cluster communication over Myrinet, Panda internally uses the LFC communication system [6]. LFC is a highly efficient, user-space communication substrate for Myrinet, similar to active messages.

For inter-cluster communication over the wide-area ATM network, Panda uses one dedicated *gateway* machine per cluster. The gateways also implement the Panda primitives, but support communication over both Myrinet and ATM. A gateway can communicate with the machines in its local cluster, using LFC over Myrinet. In addition, it can communicate with gateways of other clusters, using TCP/IP over ATM. The gateway machines thus forward traffic to and from remote clusters. In this way, the existence of multiple clusters is transparent to the Manta runtime system.

Table 1: Latency and maximum throughput of Manta and Sun JDK

	Myrinet		ATM	
	Latency (μs)	Throughput (MByte/s)	Latency (μs)	Throughput (MByte/s)
Manta	42.3	38.0	4350	0.55
Sun JDK	1228	4.66	5570	0.55

Manta's RMI protocol simply invokes Panda's communication primitives, which internally calls LFC and/or TCP/IP.

The resulting Java system thus is highly transparent, both for the programmer and the RMI implementor. The system hides several complicated issues from the programmer. For example, it uses a combination of active messages and TCP/IP, but the application programmer sees only a single communication primitive (RMI). Likewise, Java hides any differences in processor-types from the programmer.

As stated before, parallel applications often have to be aware of the structure of the wide-area system, so they can minimize communication over the wide-area links. Manta programs therefore can find out how many clusters there are and to which cluster a given machine belongs. In Section 3, we will give several examples of how this information can be used to optimize programs.

2.4 Performance measurements on the DAS system

Table 1 shows the latency and throughput obtained by Manta RMI and Sun JDK RMI over the Myrinet LAN and the ATM WAN. The latencies are measured for null-RMIs, which take zero parameters and do not return a result. The maximum throughputs are measured for RMIs that take a large array as parameter. The Manta measurements were run on the Linux operating system. With Sun JDK over ATM, we used JDK version 1.1.6 on Linux. The performance of the JDK on Myrinet was measured on BSD/OS (using the JDK 1.1.4), because we do not have a Linux port of the JDK on Myrinet yet.

For intra-cluster communication over Myrinet, Manta is much faster than the JDK, which uses a slow serialization and RMI protocol, executed using a byte code interpreter. Manta uses fast serialization routines generated by the compiler, a light-weight RMI protocol, and an efficient communication protocol (Panda). The maximum throughput of Manta is 38.0 MByte/sec. A performance breakdown of Manta RMI and JDK RMI is given in [25].

For inter-cluster communication over ATM, we used the wide area link between the DAS clusters at VU Amsterdam and TU Delft (see Figure 1), which has the longest latency (and largest distance) of the DAS wide-area links. The difference in wide-area RMI latency between Manta and the JDK is 1.2 msec. Both Manta and the JDK achieve a maximum wide-area throughput of 0.55 MByte/sec, which is almost 75% of the hardware bandwidth (6 Mbit/sec). The differences in wide-area latency between Manta and the JDK are due to Manta's more efficient serialization and RMI protocols, since both systems use the same communication layer (TCP/IP) over ATM.

3 Application experience

We implemented four parallel Java applications that communicate via RMI. Initially, the applications were designed for homogeneous (local area) networks. We adapted these single cluster versions to exploit the hierarchical structure of the wide-area system by minimizing the communication overhead over the wide area links, using optimizations similar to those described in [4, 29]. Below, we briefly discuss the original (single cluster) applications as well as the wide-area optimized programs and we give performance measurements on the DAS system. We only present results for Manta, as other competitive Java platforms (e.g., the JDK and Kaffe) are not yet available on the DAS system (using Myrinet and Linux).

For each of the four programs, we will analyze its performance on the wide-area DAS system, using the following approach. The goal of wide-area parallel programming is to obtain higher speedups on multiple clusters than on a single cluster. Therefore, we have measured the speedups of each program on a single DAS cluster and on four DAS clusters, the latter with and without wide-area optimizations. In addition, we have measured the speedups on a single cluster with the same total number of nodes, to determine how much performance is lost by using multiple distributed clusters instead of one big centralized cluster. (All speedups are computed relative to the same program on a single machine.)

The results are shown in Figure 8. The figure contains four bars for each application, giving the speedups on a single cluster of 16 nodes, four clusters of 16 nodes each (with and without wide-area aware optimizations), and a single cluster of 64 nodes. The difference between the first two bars thus indicates the performance gain by using multiple 16-node clusters (at different locations) instead of a single 16-node cluster, without any change to the application source. The performance gain achieved by the wide-area optimizations can be seen from the difference between the second and the third bar of each application. Comparing the second and the third bars with the fourth bar shows how much performance is lost (without and with wide-area optimizations) due to the slow wide-area network. (The 64-node cluster uses the fast Myrinet network between all nodes.)

3.1 Successive Overrelaxation

Red/black Successive Overrelaxation (SOR) is an iterative method for solving discretized Laplace equations on a grid. Here, it is used as an example of nearest neighbor parallelization methods. SOR is an iterative algorithm that performs multiple passes over a rectangular grid, until the grid changes less than a certain value, or a fixed number of iterations has been reached. The new value of a grid point is computed using a stencil operation, which depends only on the previous value of the point itself and its four neighbors on the grid.

The skeleton code for the single-cluster parallel Java program for SOR is given in Figure 3. The parallel algorithm we use distributes the grid row-wise among the available processors, so each machine is assigned several contiguous rows of the grid, denoted by the interval *LB* to *UB* (for lower bound and upper bound). Each processor runs a Java thread of class *SOR*, which performs the SOR iterations until the program converges (see the *run* method). Each iteration has two phases, for the red and black grid points. The processes are logically organized in a linear array. Due to the stencil operations and the row-wise distribution, every process needs one row of the grid from its left neighbor (row $LB-1$) and one row from its right neighbor (row $UB+1$). (Exceptions are made for the first and last process, but we have omitted this from our skeleton code.)

At the beginning of every iteration, each SOR thread exchanges rows with its left and right neighbors and then updates its part of the grid using this boundary information from its neighbors. The row exchange is implemented using a remote object of class *Bin* on each processor. This object is a buffer that can contain at most one row. It has synchronized methods to put and get data.

```
public remote class Bin {
    public synchronized void put(double [] row) {
        Wait until the bin is empty and save the new row.
    }

    public synchronized double [] get() {
        Wait until the bin is full and return the row.
    }
}

public remote class SOR extends RemoteThread {
    private Bin leftBin, rightBin;     // Remote bins of left and right neighbors.
    private Bin myLeftBin, myRightBin; // My own bin objects.
    private double[][] matrix;         // The matrix we are calculating on.

    public void sendRows() {
        leftBin.put(matrix[LB]);  // synchronous RMI (first row of my partition)
        rightBin.put(matrix[UB]); // synchronous RMI (last row of my partition)
    }

    public void receiveRows() {
        matrix[LB-1] = myLeftBin.get();
        matrix[UB+1] = myRightBin.get();
    }

    public void run() {
        do {    // do red/black SOR on the interval LB .. UB
          sendRows();    // Send rows LB and UB to neighbors
          receiveRows(); // Receive rows LB-1 and UB+1
          Calculate red fields in local rows LB ... UB

          sendRows();    // Send rows LB and UB to neighbors
          receiveRows(); // Receive rows LB-1 and UB+1
          Calculate black fields in local rows LB ... UB

        } while (....)
    }
}
```

Figure 3: Code skeleton for SOR, implementation for single cluster.

On a local cluster with a fast switch-based interconnect (like Myrinet), the exchange between neighbors adds little overhead, so parallel SOR obtains a high efficiency. On a wide-area system, however, the communication overhead between neighbors that are located in different clusters will be high, as such communication uses the WAN. The Java program allocates neighboring processes to the same cluster as much as possible, but the first and/or last process in each cluster will have a neighbor in a remote cluster. To hide the high latency for such inter-cluster communication, the wide-area optimized program uses split-phase communication for exchanging rows between clusters, as shown in Figure 4. It first initiates an asynchronous send for its boundary rows and then computes on the inner rows of the matrix. When this work is finished, a blocking receive

Table 2: Performance breakdown for SOR, average times in milliseconds

clusters × CPUs	optimization	speedup	total	computation	sendRows	receiveRows
1 × 16	no	15.2	76888	75697	419	772
4 × 16	no	37.5	30973	18397	2088	10488
4 × 16	yes	53.9	21601	17009	4440	152
1 × 64	no	60.6	19091	17828	754	509

for the boundary data from the neighboring machines is done, after which the boundary rows are computed.

The optimization is awkward to express in Java, since Java lacks asynchronous communication. It is implemented by using a separate thread (of class *SenderThread*) for sending the boundary data. To send a row to a process on a remote cluster, the row is first given to a newly created SenderThread; this thread will then put the row into the Bin object of the destination process on a remote cluster, using an RMI. During the RMI, the original SOR process can continue computing, so communication over the wide-area network is overlapped with computation. For communication within a cluster, the overhead of extra thread-switches slightly outweighs the benefits, so only inter-cluster communication is handled in this way (see the method *sendRows*).

The performance of the SOR program is shown in Figure 8. We ran a problem with a grid size of 4096×4096 and a fixed number of 64 iterations. The program obtains a high efficiency on a single cluster (a speedup of 60.6 on 64 processors). Without the optimization, SOR on the wide-area system achieves only a speedup of 37.5 on 4×16 processors. With the latency-hiding optimization, the speedup increases to 53.9, which is quite close to the speedup on a single 64-node cluster. Latency hiding thus is very effective for SOR. Table 2 presents a performance breakdown. It shows the total execution time in the *run* method, the time spent computing, and the time spent in the *sendRows* and *receiveRows* methods. The times in the table are the average values over all SOR threads of a run. Comparing the two runs with 4×16 CPUs shows the effectiveness of our optimization. With split-phase communication, the time spent in *receiveRows* is reduced dramatically. The price for this gain is the creation of new threads for asynchronous sending. This is why the optimized version of *sendRows* takes about twice as much time as its unoptimized (non-threaded) counterpart. In total, the split-phase communication saves about 9.5 seconds compared to the unoptimized version.

3.2 All-pairs Shortest Paths Problem

The All-pairs Shortest Paths (ASP) program finds the shortest path between any pair of nodes in a graph, using a parallel version of Floyd's algorithm. The program uses a distance matrix that is divided row-wise among the available processors. At the beginning of iteration k, all processors need the value of the kth row of the matrix. The most efficient method for expressing this communication pattern would be to let the processor containing this row (called the owner) broadcast it to all the others. Unfortunately, Java RMI does not support broadcasting, so this cannot be expressed directly in Java. Instead, we simulate the broadcast with a spanning tree algorithm implemented using RMIs and threads.

```
public class SenderThread implements Runnable {
    private Bin dest;
    private double[] row;

    public SenderThread(Bin dest, double[] row) {
        this.dest = dest;
        this.row = row;
    }

    public void run() {
        dest.put(row);
    }
}

public remote class Bin {
    // same as in single-cluster implementation
}

public remote class SOR extends RemoteThread {
    private Bin leftBin, rightBin; // Remote bins of left and right neighbors.
    private Bin myLeftBin, myRightBin; // My own bin objects.

    public void sendRows() {
        if (leftBoundary) {  // Am I at a cluster boundary?
            new SenderThread(leftBin, matrix[LB]).start(); // Asynchronous send.
        } else {
            leftBin.put(matrix[LB]); // synchronous RMI.
        }

        Same for row UB to right neighbor ...
    }

    public void receiveRows() {
        matrix[LB-1] = myLeftBin.get();
        matrix[UB+1] = myRightBin.get();
    }

    public void run() {
        do {    // do red/black SOR on the interval LB .. UB
          sendRows();    // Send rows LB and UB to neighbors
          Calculate red fields in local rows LB+1 ... UB-1
          receiveRows(); // Receive rows LB-1 and UB+1
          Calculate red fields in local rows LB and UB

          sendRows();    // Send rows LB and UB to neighbors
          Calculate black fields in local rows LB+1 ... UB-1
          receiveRows(); // Receive rows LB-1 and UB+1
          Calculate black fields in local rows LB and UB
        } while (....)
    }
}
```

Figure 4: Code skeleton for SOR, implementation for wide-area system.

The skeleton of a single-cluster implementation is shown in Figure 5. Here, all processes run a thread of class *Asp*. For broadcasting, they collectively call their *broadcast* method. Inside *broadcast*, all threads except the row owner wait until they receive the row. The owner initiates the broadcast by invoking *transfer*, which arranges all processes in a binary tree topology. Such a tree broadcast is quite efficient inside clusters with fast local networks. *transfer* sends the row to its left and right children in the tree, using daemon threads of class *Sender*. A *Sender* calls *transfer* on its destination node which recursively continues the broadcast. For high efficiency, sending inside the binary tree has to be performed asynchronously (via daemon threads) because otherwise all intermediate nodes would have to wait until the RMIs of the whole successive forwarding tree completed. As shown in Figure 8, ASP using the binary tree broadcast achieves almost linear speedup when run on a single cluster. With a graph of 2500 nodes, it obtains a speedup of 57.9 on a 64-node cluster.

A binary tree broadcast obtains poor performance on the wide-area system, causing the original ASP program to run much slower on four clusters than on a single (16-node) cluster (see Figure 8). The reason is that the spanning tree algorithm does not take the topology of the wide-area system into account, and therefore sends the same row multiple times over the same wide-area link. To overcome this problem, we implemented a wide-area optimized broadcast similar to the one used in our MagPIe collective communication library [20]. With the optimized program, the broadcast data is forwarded to all other clusters in parallel, over different wide-area links. We implement this scheme by designating one of the *Asp* processes in each cluster as a *coordinator* for that cluster. The broadcast owner asynchronously sends the rows to each *coordinator* in parallel. This is achieved by one dedicated thread of class *Sender* per cluster. Using this approach, each row is only sent once to each cluster. Due to the asynchronous send, all wide-area connections can be utilized simultaneously. Inside each cluster, a binary tree topology is used, as in the single-cluster program. The code skeleton of this implementation is shown in Figure 6. As shown in Figure 8, this optimization significantly improves ASP's application performance and makes the program run faster on four 16-node clusters than on a single 16-node cluster. Nevertheless, the speedup on four 16-node clusters lags far behind the speedup on a single cluster of 64 nodes. This is because each processor performs several broadcasts, for different iterations of the *k*-loop (see the *run* method). Subsequent broadcasts from different iterations wait for each other. A further optimization therefore would be to pipeline the broadcasts by dynamically creating new *Sender* threads (one per cluster per broadcast), instead of using dedicated daemon threads. However, this would require a large number of dynamically created threads, even increasing with the problem size. If a truly asynchronous RMI would be available, the excessive use of additional threads could be completely avoided and the overhead related to thread creation and thread switching would also disappear. Table 3 shows a performance breakdown for ASP. It shows the total execution time in the *run* method, the time spent computing, and the time spent in *broadcast*. The times in the table are the average values over all Asp threads of a run. Comparing the two runs with 4×16 CPUs, it can be seen that the wide-area optimization saves most of the communication costs of the unoptimized version. But even with the optimization, wide-area communication of 4×16 CPUs takes much more time, compared to 64 CPUs in a single cluster. By comparing the computation times of the three configurations with 64 CPUs it becomes obvious that the inferior speedup of the 4×16 configurations is due to slow wide-area communication.

3.3 The Traveling Salesperson Problem

The Traveling Salesperson Problem (TSP) computes the shortest path for a salesperson to visit all cities in a given set exactly once, starting in one specific city. We use a branch-and-bound algorithm, which prunes a large part of the search space by ignoring partial routes that are already

```
class Sender extends Thread {
    Asp dest;
    int[] row;
    int k, owner
    boolean filled = false;

    synchronized void put(Asp dest, int[] row, int k, int owner) {
        while(filled) wait();
        this.dest = dest;
        this.row = row;
        this.k = k;
        this.owner = owner;
        filled = true;
        notifyAll();
    }

    synchronized void send() {
        while(!filled) wait();
        dest.transfer(row, k, owner);  // do RMI to a child
        filled = false;
        notifyAll();
    }

    public void run() {
        while(true) send();
    }
}

public remote class Asp extends RemoteThread {
    private int[][] tab;     // The distance table.
    private Asp left, right; // My left and right successors (children) in the broadcast tree.
    private Sender leftSender, rightSender; // Threads that will do the RMI to my children.

    public synchronized void transfer(int[] row, int k, int owner) {
        if(left != null) leftSender.put(left, row, k, owner);
        if(right != null) rightSender.put(right, row, k, owner);
        tab[k] = row;
        notifyAll();  // wake up thread waiting for row to arrive
    }

    public synchronized void broadcast(int k, int owner) {
        if(this cpu is the owner) transfer(tab[k], k, owner);
        else while (tab[k] == null) wait();  // wait until the row has arrived
    }

    public void run() {  // computation part
        int i, j, k;

        for (k = 0; k < n; k++) {
            broadcast(k, owner(k));
            for (i = LB; i < UB; i++) // recompute my rows
                if (i != k)
                for (j = 0; j < n; j++)
                    tab[i][j] = minimum(tab[i][j], tab[i][k] + tab[k][j]);
        }
    }
}
```

Figure 5: Code skeleton for ASP, implementation for single cluster.

longer than the current best solution. The program is parallelized by distributing the search space

```
class Sender extends Thread {
        // Same as in single-cluster implementation.
}

public remote class Asp extends RemoteThread {
    private int[][] tab;              // The distance table.
    private Asp[] coordinators;       // The remote cluster coordinators.
    private Sender[] waSenders;       // Threads that will do the wide-area send.

    public synchronized void transfer(int[] row, int k, int owner) {
        // Same as in single-cluster implementation.
    }

    public synchronized void broadcast(int k, int owner) {
        if(this cpu is the owner) {
            // Use a separate daemon thread to send to each cluster coordinator.
            for(int i=0; i<nrClusters; i++) {
                    waSenders[i].put(coordinators[i], row, k, owner);
            }
        } else {
            while (tab[k] == null) wait();
        }
    }

    public void run() {  // computation part
        // Same as in single-cluster implementation.
    }
}
```

Figure 6: Code skeleton for ASP, implementation for wide-area system.

over the different processors. Because the algorithm performs pruning, however, the amount of computation needed for each sub-space is not known in advance and varies between different parts of the search space. Therefore, load balancing becomes an issue. In single-cluster systems, load imbalance can easily be minimized using a centralized job queue. In a wide-area system, this would also generate much wide-area communication. As wide-area optimization we implemented one job queue per cluster. The work is initially equally distributed over the queues. Job stealing between the cluster queues balances the load during runtime without excessive wide-area communication. The job queues are *remote* objects, so they can be accessed over the network using RMI. Each job contains an initial path of a fixed number of cities; a processor that executes the job computes the lengths of all possible continuations, pruning paths that are longer than the current best solution. Each processor runs one *worker* thread that repeatedly fetches jobs from the job queue of its cluster

Table 3: Performance breakdown for ASP, average times in milliseconds

clusters × CPUs	optimization	speedup	total	computation	broadcast
1 × 16	no	15.6	230173	227908	2265
4 × 16	no	2.5	1441045	57964	1383081
4 × 16	yes	24.3	147943	56971	90972
1 × 64	no	57.9	61854	57878	3976

(using RMI) and executes the job, until all work is finished.

The TSP program keeps track of the current best solution found so far, which is used to prune part of the search space. Each worker contains a copy of this value in an object of class *Minimum*. If a worker finds a better complete route, the program does an RMI to all other peer workers to update their copies. To allow these RMIs, the *Minimum* values are declared as remote objects. The implementation shown in Figure 7 is rather straight forward and will not scale to very large numbers of workers. In that case, a (cluster-aware) forwarding tree would perform better. Fortunately, these updates happen infrequently. Using a 17-city problem, we counted as few as 7 updates during the whole run of 290 seconds when using 64 CPUs.

The performance for the TSP program on the wide-area DAS system is shown in Figure 8, using a 17-city problem. The runtime of our TSP program is strongly influenced by the actual job distribution which results in different execution orders and hence in different amounts of routes that can be pruned. To avoid this problem, the results presented in Figure 8 and Table 4 have been obtained by initializing the *Minimum* value to the length of the resulting shortest path. Consequently, the *Minimum* objects are never updated, but pruning is always the same, independent of configuration and execution order. Runtime differences are thus only due to communication behavior.

As can be seen in Figure 8, the speedup of TSP on the wide-area system is only slightly inferior than on a single 64-node cluster. Our wide-area optimized version is even slightly faster than the unoptimized version on a single, large cluster. This is presumably because there is less contention on the queue objects when the workers are distributed over four queues. To verify this assumption, we also used our optimized version with four queues with 1×64 CPUs and obtained another slight performance improvement compared to a single queue and 1×64 CPUs. Table 4 supports this presumption. It presents the speedups, the average time spent by the worker threads in total, while computing, and while getting new jobs. It also shows the average number of jobs processed per worker and the average time per get operation. The speedups are computed for the whole parallel application (the maximum time over all processes), while the time values in the table are averaged over the worker threads.

On a single cluster, the average time per get operation with 64 CPUs is three times as high as with only 16 CPUs. This fact supports the assumption that contention at the queue-owning CPUs is a problem. Unfortunately, we were not able to directly measure the time spent serving incoming get requests. The total time spent in the *get* method is very low compared to the computation time which explains the high speedup values. With four queues, the average *get* time is much higher than with a single queue, because job stealing between queues takes additional time. But as the achieved speedups suggest, the benefits of having multiple queues more than outweigh the additional costs with getting jobs in this case.

3.4 Iterative Deepening A*

Iterative Deepening A* is another combinatorial search algorithm, based on repeated depth-first searches. IDA* tries to find a solution to a given problem by doing a depth-first search up to a certain maximum depth. If the search fails, it is repeated with a higher search depth, until a solution is found. The search depth is initialized to a lower bound of the solution. The algorithm thus performs repeated depth-first searches. Like branch-and-bound, IDA* uses pruning to avoid

```
public remote class Minimum {
   private int minimum;  // the minimum value itself
   private Minimum[] PeerTable;   // table of peers

   public synchronized void update(int minimum){
      if (minimum < this.minimum) {
         this.minimum = minimum;
   }

   public synchronized void set(int minimum) {
      if (minimum < this.minimum) {
         this.minimum = minimum;

         // notify all peers
         for (int i=0; i < PeerTable.length; i++) {
            PeerTable[i].update(minimum);
         }
      }
   }
}
```

Figure 7: Code skeleton for TSP, update of current best solution.

Table 4: Performance breakdown for TSP, average times in milliseconds

clusters × CPUs	optimization	speedup	time total	time computation	time get	get operations number	get operations time per get
1 × 16	no	16.0	509768	509696	72	211	0.3
4 × 16	no	60.7	128173	127904	269	53	5.1
4 × 16	yes	61.1	129965	127270	2695	53	50.9
1 × 64	no	60.9	128144	128094	50	53	0.9
1 × 64	yes	61.7	131763	127184	4579	53	86.4

searching useless branches.

We have written a parallel IDA* program in Java for solving the 15-puzzle (the sliding tile puzzle). IDA* is parallelized by searching different parts of the search tree concurrently. The program uses a more advanced load balancing mechanism than TSP, based on work stealing. Each machine maintains its own job queue, but machines can get work from other machines when they run out of jobs. Each job represents a node in the search space. When a machine has obtained a job, it first checks whether it can prune the node. If not, it expands the node by computing the successor states (children) and stores these in its local job queue. To obtain a job, each machine first looks in its own job queue; if it is empty it tries the job queues of some other, randomly selected machines. We implemented one wide-area optimization: to avoid wide-area communication for work stealing, each machine first tries to steal jobs from machines in its own cluster. Only if that fails, the work queues of remote clusters are accessed. In each case, the same mechanism (RMI) is used to fetch work, so this heuristic is easy to express in Java.

Figure 8 shows the speedups for the IDA* program. The program takes about 5% longer on the wide-area DAS system than on a single cluster with 64 nodes. The communication overhead is due to work-stealing between clusters and to the distributed termination detection algorithm

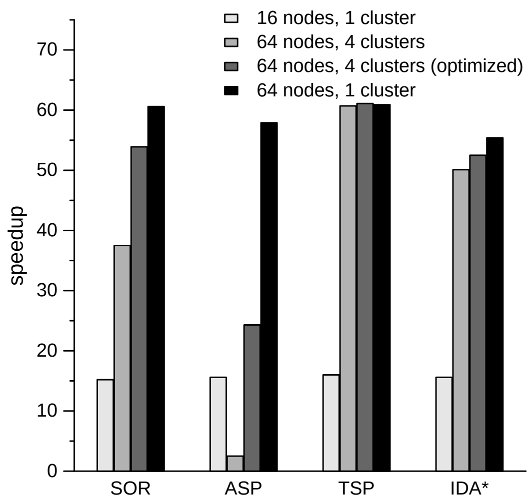

Figure 8: Speedups of four Java applications on a single cluster of 16 nodes, 4 WAN-connected clusters of 16 nodes (original and optimized program), and a single cluster of 64 nodes.

Table 5: Performance breakdown for IDA*, average times in milliseconds

				time			jobs stolen	
clusters × CPUs	optimization	speedup	total	computation	get	local	remote
1 × 16	no	15.6	77384	75675	1709	69	
4 × 16	no	50.1	23925	19114	4811	46	15
4 × 16	yes	52.5	22782	19098	3684	62	8
1 × 64	no	55.4	21795	19107	2688	70	

used by the program. The gain of the wide-area optimization is small in this case. For obtaining meaningful results across various parallel configurations, our IDA* implementation searches all solutions of equal minimal depth for a given puzzle. Table 5 shows a performance breakdown for IDA*. The times presented are averages over all threads, showing the times spent in total, while computing, and while getting new jobs. Comparing the two runs with 4×16 CPUs it can be seen that our wide-area optimization is effective, but has only a minor impact on the total completion time. This is because job stealing occurs infrequently. The numbers of actually stolen jobs shown in the table are also average values over all threads. It can be seen that the optimization helps reducing wide-area communication by reducing the number of jobs stolen from remote clusters.

4 Alternative programming models

We have discussed the implementation and wide-area optimization of four parallel applications using the RMI model. RMI supports transparent invocation of methods on remote objects and thus is a natural extension of Java's object model to distributed memory systems. In this section, we discuss alternative programming models and we compare them to RMI in terms of expressiveness (ease of programming) and implementation efficiency. We will discuss replicated objects [3, 23, 26, 33], JavaSpaces [15], and MPI for Java [10]. These alternative models have not been implemented in the Manta system, so we do not provide application measurements. Our experiences in wide-area parallel programming using replicated objects and MPI in combination with other languages (Orca and C) are described elsewhere[4, 20, 29]. We first briefly describe the four programming models.

4.1 The programming models

RMI. With RMI, parallel applications strictly follow Java's object-oriented model in which client objects invoke methods on server objects in a location-transparent way. Each remote object is physically located at one machine. Although the RMI model hides object remoteness from the programmer, the actual object location strongly impacts application performance.

Replication. From the client's point of view, object replication is conceptually equivalent to the RMI model. The difference is in the implementation: objects may be physically replicated on multiple processes. The advantage is that read-only operations can be performed locally, without any communication. The disadvantage is that write operations become more complex and have to keep object replicas consistent.

JavaSpaces. JavaSpaces adapt the Linda model [16] to the Java language. Communication occurs via shared data *spaces* into which *entries* (typed collections of Java objects) may be written. Inside a space, entries may not be modified, but they may be read or removed (taken) from a space. A reader of an entry provides a *template* that matches the desired entry type and also desired object values stored in the entry. Wildcards may be used for object values. Additionally, a space may notify an object whenever an entry matching a certain template has been written. *Space* objects may be seen as objects that are remote to all communicating processes; read and write operations are implemented as RMIs to space objects. JavaSpaces also supports a transaction model, allowing multiple operations on space objects to be combined in a transaction that either succeeds or fails as a whole. This feature is especially useful for fault-tolerant programs.

MPI. With the Message Passing Interface (MPI) language binding to Java, communication is expressed using message passing rather than remote method invocations. Processes send messages (arrays of objects) to each other. Additionally, MPI defines collective operations in which all members of a process group collectively participate; examples are broadcast and related data redistributions, reduction computations (e.g., computing global sums), and barrier synchronization.

4.2 Comparison of the models

To compare the suitability of the models for wide-area parallel computing, we study three important aspects. Table 6 summarizes this comparison. In general, we assume that the underlying programming system (like Manta) exposes the physical distribution of the clustered wide-area system.

This information may be used either by application programs (as with Manta) or by programming platforms designed for wide-area systems (e.g., the MagPIe library [20]).

Synchronous vs. asynchronous communication. Remote method invocation is a typical example of synchronous communication. Here, the client has to wait until the server object has returned the result of the invoked method. This enforces rendezvous-style synchronization of the client with another, possibly remote process. With asynchronous communication, the client may immediately continue its operation after the communication has been initiated. It may later check or wait for completion of the communication operation. Asynchronous communication is especially important for wide-area computing, where it can be used to hide the high message latencies by overlapping communication and computation.

MPI provides asynchronous sending and receiving. The other three models, however, rely on synchronous method invocation so applications have to simulate asynchronous communication using multithreading. For local-area communication, the corresponding overhead for thread creation and context switching may exceed the cost of a synchronous RMI. To cope with this problem, the optimized code for SOR gets rather complicated, as has been shown in the previous section. The broadcast implementation in ASP also requires asynchronous communication, both in the original (single cluster) and wide-area optimized version. (With synchronous RMI, a broadcast sender would have to wait for the whole spanning tree to complete.) The TSP code could also be improved by treating local and remote communication differently when updating the value of the current best solution. Fortunately, TSP is less sensitive to this problem because these updates occur infrequently. In IDA*, work stealing is always synchronous but fortunately infrequent. So, IDA* is hardly affected by synchronous RMI.

In conclusion, for wide-area parallel computing on hierarchical systems, directly supporting asynchronous communication (as in MPI) is easier to use and more efficient than using synchronous communication and multithreading.

Explicit vs. implicit receipt. A second issue is the way in which messages are received and method invocations are served. With RMI and with replicated objects, method invocations cause upcalls on the server side, which is a form of implicit message receipt. MPI provides only explicit message receipt, making the implementations of TSP and IDA* much harder. Here, incoming messages (for updating the global bound or for job requests) have to be polled for by the applications. This complicates the programs and makes them also less efficient, because finding the right polling frequency (and the best places in the applications to put polling statements at) is difficult [3]. With clustered wide-area systems, polling has to simultaneously satisfy the needs of LAN and WAN communication, making the problem of finding the right polling frequency even harder. With programming models like RMI that support implicit receipt, application-level polling is not necessary.

JavaSpaces provide explicit operations for reading and removing entries from a space, resulting in the same problems as with MPI. Additionally, a space object may notify a potential reader whenever an entry has been written that matches the reader's interests. Unfortunately, this notification simply causes an upcall at the receiver side which in turn has to actually perform the read or take operation. This causes a synchronous RMI back to the space object. The additional overhead can easily outweigh the benefit of implicit receipt, especially when wide-area networks are used.

Table 6: Aspects of programming models

	RMI	replication	JavaSpaces	MPI
send	synchronous	synchronous	synchronous	synchronous and asynchronous
receive	implicit	implicit	explicit and implicit	explicit
collective communication	no	broadcast	no	yes

Point-to-point vs. collective communication. Wide-area parallel programming systems can ease the task of the programmer by offering higher-level primitives that are mapped easily onto the hierarchical structure of the wide-area system. In this respect, we found MPI's collective operations (e.g., broadcast and reduction) to be of great value. The MagPIe library [20] optimizes MPI's collective operations for clustered wide-area systems by exploiting knowledge about how groups of processes interact. For example, a broadcast operation defines data transfers to all processes of a group. MagPIe uses this information to implement a broadcast that optimally utilizes wide-area links; e.g., it takes care that data is sent only once to each cluster. Replicated objects that are implemented with a write-update protocol [3] can use broadcasting for write operations, and thus can also benefit from wide-area optimized broadcast.

Programming models without group communication (like RMI and JavaSpaces) cannot provide such wide-area optimizations inside a runtime system. Here, the optimizations are left to the application itself, making it more complex. The broadcast implemented for ASP is an example of such an application-level optimization that could be avoided by having group communication in the programming model. The MagPIe implementation of ASP [20], for example, is much simpler than the (wide-area optimized) Java version. Similarly, TSP's global bound could be updated using a pre-optimized group operation.

4.3 Summary

The remote method invocation model provides a good starting point for wide-area parallel programming, because it integrates communication into the object model. Another benefit is RMI's implicit receipt capability by which RMIs are served using upcalls. This contributes to expressiveness of the programming model as well as to program efficiency, especially with wide-area systems.

To be fully suited for parallel systems, RMI needs two augmentations. The first one is asynchronous method invocation, especially important with wide-area systems. The second extension is the implementation of collective communication operations that transfer the benefits of MPI's collective operations into Java's object model.

Although promising at first glance, JavaSpaces does not really make wide-area parallel programming easier, because operations on space objects are neither asynchronous nor collective. As fault-tolerance becomes more important in wide-area computing, the JavaSpaces model along with its transaction feature may receive more attention.

5 Related work

We have discussed a Java-centric approach to writing wide-area parallel (metacomputing) applications. Most other metacomputing systems (e.g., Globus [13] and Legion [17]) support a variety of languages. The SuperWeb [1], the Gateway system [18], Javelin [11], Javelin++ [9], and Bayanihan [31] are examples of global computing infrastructures that support Java. A language-centric approach makes it easier to deal with heterogeneous systems, since the data types that are transferred over the networks are limited to the ones supported in the language (thus obviating the need for a separate interface definition language) [36].

Most work on metacomputing focuses on how to build the necessary infrastructure [2, 13, 17, 30]. In addition, research on parallel algorithms and applications is required, since the bandwidth and latency differences in a metacomputer can easily exceed three orders of magnitude [12, 13, 17, 29]. Coping with such a large non-uniformity in the interconnect complicates application development. The ECO system addresses this problem by automatically generating optimized communication patterns for collective operations on heterogeneous networks [24]. The AppLeS project favors the integration of workload scheduling into the application level [5].

In our earlier research, we experimented with optimizing parallel programs for a hierarchical interconnect, by changing the communication structure [4]. Also, we studied the sensitivity of such optimized programs to large differences in latency and bandwidth between the LAN and WAN [29]. Based on this experience, we implemented collective communication operations as defined by the MPI standard, resulting in improved application performance on wide area systems [20]. Some of the ideas of this earlier work have been applied in our wide-area Java programs.

There are many other research projects for parallel programming in Java. Titanium [37] is a Java-based language for high-performance parallel scientific computing. The JavaParty system [28] is designed to ease parallel cluster programming in Java. In particular, its goal is to run multi-threaded programs with as little change as possible on a workstation cluster. Hyperion [26] also uses the standard Java thread model as a basis for parallel programming. Unlike JavaParty, Hyperion caches remote objects to improve performance. The Do! project tries to ease parallel programming in Java using parallel and distributed frameworks [22]. Ajents [19] is a parallel programming environment that supports object migration. Java/DSM [38] implements a JVM on top of a distributed shared memory system. Breg et al. [8] study RMI performance and interoperability. Krishnaswamy et al. [21] improve RMI performance somewhat with caching and by using UDP instead of TCP. Nester et al. [27] present new RMI and serialization packages (drop-in replacements) designed to improve RMI performance. The above Java systems are designed for single-level ("flat") parallel machines. The Manta system described in this paper, on the other hand, is designed for hierarchical systems and uses different communication protocols for local and wide area networks. It uses a highly optimized RMI implementation, which is particularly effective for local communication.

6 Conclusions

We have described our experiences in building and using a high-performance Java system that runs on a geographically distributed (wide-area) system. The goal of our work was to obtain actual experience with a Java-centric approach to metacomputing. Java's support for parallel processing and heterogeneity make it an attractive candidate for metacomputing. The Java system we have built, for example, is highly transparent: it provides a single communication primitive (RMI) to the user, even though the implementation uses several communication networks and protocols.

Our Manta programming system is designed for hierarchical wide-area systems, for example clusters or MPPs connected by wide-area networks. Manta uses a very efficient (active message like) communication protocol for the local interconnect (Myrinet) and TCP/IP for wide-area communication. The two communication protocols are provided by the Panda library. Manta's lightweight RMI protocol is implemented on top of Panda and is very efficient.

We have implemented several parallel applications on this system, using Java RMI for communication. In general, the RMI model was easy to use. To obtain good performance, the programs take the hierarchical structure of the wide-area system into account and minimize the amount of communication (RMIs) over the slow wide-area links. With such optimizations in place, the programs can effectively use multiple clusters, even though they are connected by slow links.

We compared RMI with other programming models, namely object replication, JavaSpaces, and MPI for Java. We identified several shortcomings of the RMI model. In particular, the lack of asynchronous communication and broadcast complicates programming. MPI offers both features (and further useful collective operations) but lacks RMI's clean object-oriented model. Object replication is closer to pure RMI but offers only broadcast-like object updating. A careful integration of these features into an RMI-based system is our next goal for a Java-centric programming platform for wide-area parallel programming.

Acknowledgements

This work is supported in part by a SION grant from the Dutch research council NWO, and by a USF grant from the Vrije Universiteit. The wide-area DAS system is an initiative of the Advanced School for Computing and Imaging (ASCI). We thank Raoul Bhoedjang, Rutger Hofman, Ceriel Jacobs, Aske Plaat, and Cees Verstoep for their contributions to this research. We thank John Romein and Cees Verstoep for keeping the DAS in good shape, and Cees de Laat (University of Utrecht) for getting the wide area links of the DAS up and running.

References

[1] A. D. Alexandrov, M. Ibel, K. E. Schauser, and C. J. Scheiman. SuperWeb: Research Issues in Java-Based Global Computing. *Concurrency: Practice and Experience*, 9(6):535–553, June 1997.

[2] J. Arabe, A. Beguelin, B. Lowekamp, E. Seligman, M. Starkey, and P. Stephan. Dome: Parallel programming in a heterogeneous multi-user environment. In *10th International Parallel Processing Symposium*, pages 218–224, Honolulu, Hawaii, Apr. 1996.

[3] H. E. Bal, R. Bhoedjang, R. Hofman, C. Jacobs, K. Langendoen, T. Rühl, and F. Kaashoek. Performance Evaluation of the Orca Shared Object System. *ACM Transactions on Computer Systems*, 16(1):1–40, Feb. 1998.

[4] H. E. Bal, A. Plaat, M. G. Bakker, P. Dozy, and R. F. Hofman. Optimizing Parallel Applications for Wide-Area Clusters. In *12th International Parallel Processing Symposium (IPPS'98)*, pages 784–790, Orlando, FL, April 1998.

[5] F. Berman, R. Wolski, S. Figueira, J. Schopf, and G. Shao. Application-level Scheduling on Distributed Heterogeneous Networks. In *Supercomputing 96*, Pittsburgh, PA, Nov. 1996.

[6] R. A. F. Bhoedjang, T. Rühl, and H. E. Bal. User-Level Network Interface Protocols. *IEEE Computer*, 31(11):53–60, Nov. 1998.

[7] N. Boden, D. Cohen, R. Felderman, A. Kulawik, C. Seitz, J. Seizovic, and W. Su. Myrinet: A Gigabit-per-second Local Area Network. *IEEE Micro*, 15(1):29–36, Feb. 1995.

[8] F. Breg, S. Diwan, J. Villacis, J. Balasubramanian, E. Akman, and D. Gannon. Java RMI Performance and Object Model Interoperability: Experiments with Java/HPC++ Distributed Components. In *ACM 1998 Workshop on Java for High-Performance Network Computing*, Santa Barbara, CA, Feb. 1998.

[9] S. Brydon, P. Kmiec, M. Neary, S. Rollins, and P. Cappello. Javelin++: Scalability Issues in Global Computing. In *ACM 1999 Java Grande Conference*, pages 171–180, San Francisco, CA, June 1999.

[10] B. Carpenter, V. Getov, G. Judd, T. Skjellum, and G. Fox. MPI for Java: Position Document and Draft API Specification. Technical Report JGF-TR-03, Java Grande Forum, November 1998.

[11] B. Christiansen, P. Cappello, M. F. Ionescu, M. O. Neary, K. E. Schauser, and D. Wu. Javelin: Internet-Based Parallel Computing Using Java. *Concurrency: Practice and Experience*, 1997.

[12] I. Foster, J. Geisler, W. Gropp, N. Karonis, E. Lusk, G. Thiruvathukal, and S. Tuecke. Wide-Area Implementation of the Message Passing Interface. *Parallel Computing*, 24(12–13):1735–1749, 1998.

[13] I. Foster and C. Kesselman. Globus: A Metacomputing Infrastructure Toolkit. *International Journal of Supercomputer Applications*, 11(2):115–128, Summer 1997.

[14] I. Foster and C. Kesselman, editors. *The GRID: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann, 1998.

[15] E. Freeman, S. Hupfer, and K. Arnold. *JavaSpaces, Principles, Patterns, and Practice*. Addison Wesley, 1999.

[16] D. Gelernter. Generative Communication in Linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, 1985.

[17] A. Grimshaw and W. A. Wulf. The Legion Vision of a Worldwide Virtual Computer. *Communications of the ACM*, 40(1):39–45, Jan. 1997.

[18] T. Haupt, E. Akarsu, G. Fox, A. Kalinichenko, K.-S. Kim, P. Sheethalnath, and C.-H. Youn. The Gateway System: Uniform Web Based Access to Remote Resources. In *ACM 1999 Java Grande Conference*, pages 1–7, San Francisco, CA, June 1999.

[19] M. Izatt, P. Chan, and T. Brecht. Ajents: Towards an Environment for Parallel, Distributed and Mobile Java Applications. In *ACM 1999 Java Grande Conference*, pages 15–24, San Francisco, CA, June 1999.

[20] T. Kielmann, R. F. H. Hofman, H. E. Bal, A. Plaat, and R. A. F. Bhoedjang. MAGPIE: MPI's Collective Communication Operations for Clustered Wide Area Systems. In *Seventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'99)*, pages 131–140, Atlanta, GA, May 1999.

[21] V. Krishnaswamy, D. Walther, S. Bhola, E. Bommaiah, G. Riley, B. Topol, and M. Ahamad. Efficient Implementations of Java RMI. In *4th USENIX Conference on Object-Oriented Technologies and Systems (COOTS'98)*, Santa Fe, NM, 1998.

[22] P. Launay and J.-L. Pazat. The Do! project: Distributed Programming Using Java. In *First UK Workshop Java for High Performance Network Computing*, Southampton, Sept. 1998.

[23] S. Y. Lee. *Supporting Guarded and Nested Atomic Actions in Distributed Objects*. Master's thesis, University of California at Santa Barbara, July 1998.

[24] B. Lowekamp and A. Beguelin. ECO: Efficient Collective Operations for Communication on Heterogeneous Networks. In *International Parallel Processing Symposium*, pages 399–405, Honolulu, HI, 1996.

[25] J. Maassen, R. van Nieuwpoort, R. Veldema, H. E. Bal, and A. Plaat. An Efficient Implementation of Java's Remote Method Invocation. In *Seventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'99)*, pages 173–182, Atlanta, GA, May 1999.

[26] M. W. Macbeth, K. A. McGuigan, and P. J. Hatcher. Executing Java Threads in Parallel in a Distributed-Memory Environment. In *Proc. CASCON'98*, pages 40–54, Missisauga, ON, 1998. Published by IBM Canada and the National Research Council of Canada.

[27] C. Nester, M. Philippsen, and B. Haumacher. A More Efficient RMI for Java. In *ACM 1999 Java Grande Conference*, pages 153–159, San Francisco, CA, June 1999.

[28] M. Philippsen and M. Zenger. JavaParty—Transparent Remote Objects in Java. *Concurrency: Practice and Experience*, pages 1225–1242, Nov. 1997.

[29] A. Plaat, H. E. Bal, and R. F. H. Hofman. Sensitivity of Parallel Applications to Large Differences in Bandwidth and Latency in Two-Layer Interconnects. In *High Performance Computer Architecture (HPCA-5)*, pages 244–253, Orlando, FL, January 1999.

[30] A. Reinefeld, R. Baraglia, T. Decker, J. Gehring, D. Laforenza, J. Simon, T. Rümke, and F. Ramme. The MOL Project: An Open Extensible Metacomputer. In *Heterogenous computing workshop HCW'97 at IPPS'97*, Apr. 1997.

[31] L. F. G. Sarmenta and S. Hirano. Bayanihan: Building and Studying Web-Based Volunteer Computing Systems Using Java. *Future Generation Computer Systems*, 15(5/6), 1999.

[32] L. Smarr and C. Catlett. Metacomputing. *Communications of the ACM*, 35(6):44–52, June 1992.

[33] B. Topol, M. Ahamad, and J. T. Stasko. Robust State Sharing for Wide Area Distributed Applications. In *18th International Conference on Distributed Computing Systems (ICDCS'98)*, pages 554–561, Amsterdam, The Netherlands, May 1998.

[34] R. van Nieuwpoort, J. Maassen, H. E. Bal, T. Kielmann, and R. Veldema. Wide-area parallel computing in Java. In *ACM 1999 Java Grande Conference*, pages 8–14, San Francisco, CA, June 1999.

[35] J. Waldo. Remote procedure calls and Java Remote Method Invocation. *IEEE Concurrency*, pages 5–7, July–September 1998.

[36] A. Wollrath, J. Waldo, and R. Riggs. Java-Centric Distributed Computing. *IEEE Micro*, 17(3):44–53, May/June 1997.

[37] K. Yelick, L. Semenzato, G. Pike, C. Miyamoto, B. Liblit, A. Krishnamurthy, P. Hilfinger, S. Graham, D. Gay, P. Colella, and A. Aiken. Titanium: a High-performance Java Dialect. In *ACM 1998 Workshop on Java for High-Performance Network Computing*, Feb. 1998.

[38] W. Yu and A. Cox. Java/DSM: A Platform for Heterogeneous Computing. *Concurrency: Practice and Experience*, pages 1213–1224, Nov. 1997.