Wide-area parallel computing in Java

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

Dept. of Mathematics and Computer Science,
Vrije Universiteit, Amsterdam, The Netherlands

Keywords: metacomputing, remote method invocation, communication protocols, high-speed networks, DAS

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 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.

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 [11, 21]. 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 [22], 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, it is interesting for several reasons. The system, called *Manta*, has been optimized to achieve high performance. 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 [16]. 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 44 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.

In this paper, 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. This experience will be useful for future (Java-centric) metacomputing systems and may also stimulate further research on programming support for wide-area applications. The outline of the paper is as follows. In

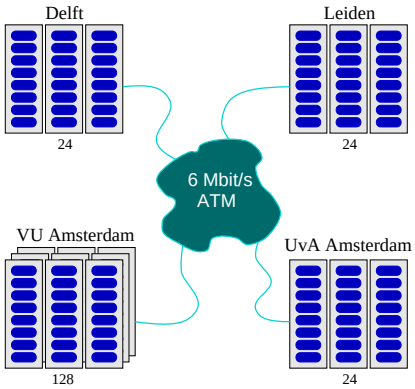

Figure 1: The wide-area DAS system.

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 look at related work and in Section 5 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 1.1.4 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 [6]. 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 BSD/OS 3.0. 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 [19] (and on *http://www.cs.vu.nl/das/*).

2.2 The Manta system

Manta is a Java system designed for high-performance parallel computing. Like JavaParty [18], Manta uses a separate `remote` keyword to indicate which classes allow their methods to be invoked remotely, which is somewhat more flexible and easier to use than the standard RMI mechanism (i.e., inheriting from class `java.rmi.server.UnicastRemoteObject`). Except for this, 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 (de)serialization 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 lightweight RMI protocol. Additionally, Manta is designed to allow interoperability with other JVMs, using Sun's RMI protocol and a technique to dynamically compile and link byte codes received from other JVMs. Below, we briefly discuss Manta's basic RMI protocol; for more details on the protocol and on the interoperability with other JVMs, we refer to [16].

The Manta RMI protocol was 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.

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 inspection. 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 [16].

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 [5]. 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

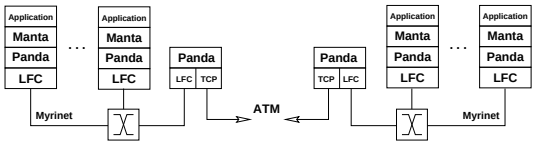

Figure 2: Wide area communication based on Panda

clusters is transparent to the Manta runtime system. 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.[1]

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.[2] 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.

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 (compiler-generated) serialization routines, a light-weight RMI protocol, and an efficient communication protocol (Panda). The maximum throughput of Manta (achieved with arrays of size 16 KByte) is 24.7 MByte/sec. A performance

[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.

[2] The latency of Manta RMI over Myrinet is somewhat higher than that reported in an earlier paper [16]. This is due to a difference in the Panda layer, which now supports a scatter/gather interface but has a higher overhead for small messages.

	Myrinet		ATM	
	Latency (μs)	Throughput (MByte/s)	Latency (μs)	Throughput (MByte/s)
Manta	44	24.7	5600	0.55
Sun JDK 1.1.4	1228	4.66	6700	0.35

Table 1: Latency and maximum throughput of Manta and Sun JDK 1.1.4

breakdown of Manta RMI and JDK RMI is given in [16].

For inter-cluster communication over ATM, we used the wide area link between the DAS clusters at VU Amsterdam an TU Delft (see Fig. 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 1.1 msec. Manta achieves a maximum wide-area throughput of 0.55 MByte/sec, which is almost 75% of the hardware bandwidth (6 Mbit/sec). The JDK RMI protocol is able to achieve less than 50% of the wide-area bandwidth. The differences in wide-area latency and throughput 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. Most applications exploit the hierarchical structure of the wide-area system to minimize the communication overhead over the wide area links, using optimizations similar to those described in [4, 19]. Below, we briefly discuss the optimized applications and we give performance measurements on the DAS system. We only present results for Manta, as other competitive Java platforms are not yet available for the DAS system. (Sun's JDK 1.1.4 uses only interpreted code, Sun's JIT 1.1.6 is not available for BSD/OS, and the Kaffe JIT does not yet support RMI.)

3.1 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 longer than the current best solution. The program is parallelized by distributing the search space 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. Our Java program performs load balancing using a centralized FIFO job queue. The job queue is a *remote* object, so it can be accessed over the (local or wide-area) network using RMI. One manager process generates all jobs (partial routes) and adds them to the queue. 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* process, which is a remote object containing an active thread. The worker repeatedly fetches jobs from the job queue (using RMI) and executes it, 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. If a worker finds a better complete route, the program does an RMI to all other workers to update their copies. (To allow these RMIs, the workers are declared as remote objects.)

The performance for the TSP program on the wide-area DAS system is shown in Figure 3, using a 17-city problem. The figure shows the speedups of the program on 4 clusters of 10 machines, and compares it with the performance on a single cluster of 10 nodes and a single cluster of 40 nodes. (All speedups are computed relative to the same program on a single machine.) The difference between the first two bars indicates the performance gain by using multiple 10-node clusters (at different locations) instead of a single 10-node cluster. The difference between the second and third bar shows how much performance is lost due to the slow wide-area network (since the 40-node cluster uses the fast Myrinet network between all nodes). As can be seen, the performance of TSP on the wide-area system is the same as that on a local cluster with the same number of processors. The reason is that the program is rather coarse-grained.

3.2 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 searching useless branches.

We have written a parallel IDA* program in Java for solving the 15-puzzle (the sliding tile puzzle). IDA* is

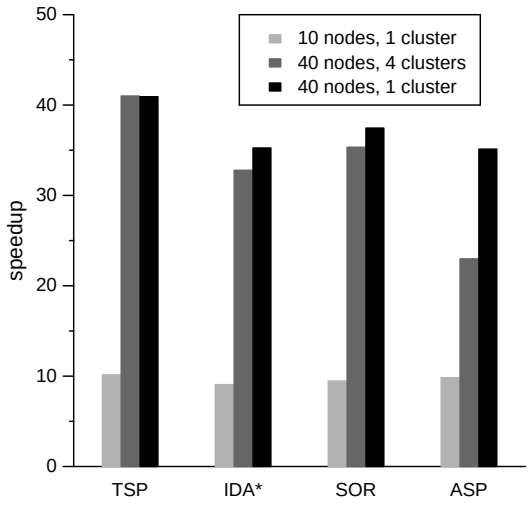

Figure 3: Speedups of four Java applications on a single cluster of 10 nodes, 4 clusters of 10 nodes each connected by a WAN, and a single cluster of 40 nodes.

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 nodes 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 machines. 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 3 shows the speedups for the IDA* program. The program takes about 7% longer on the wide-area DAS system than on a single cluster with 40 nodes. The overhead is due to the work-stealing between clusters.

3.3 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. The parallel algorithm we use distributes the grid row-wise among the available processors, which are logically organized in a linear array. At the beginning of every iteration, each processor (except the first and last one) exchanges rows with its left and right neighbors and then updates its part of the grid using this boundary information from its neighbors.

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 program uses split-phase communication for exchanging rows between clusters. It first initiates an asynchronous send for the boundary rows and then computes on the inner rows of the matrix. When this work is finished, a blocking receive for the boundary data from the neighboring machines is done, after which the boundary rows are recomputed. The optimization is awkward to express in Java, since Java lacks asynchronous communication. It is implemented by using a separate thread for sending the boundary data. For communication within a cluster, however, the overhead of extra thread-switches outweighs the benefits, so only inter-cluster communication is handled in this way.

The performance of the SOR program is shown in Figure 3. Due to the latency-hiding optimization, the program is only 6% slower on the wide-area system than on a local 40-node cluster.

3.4 All-pairs Shortest Paths Problem

The goal of the All-pairs Shortest Paths (ASP) problem is to find the shortest path between any pair of nodes in a graph. The program uses a distance matrix that is divided row-wise among the available processors. At the beginning of iteration i, all processors need the value of the ith row of the matrix. The most efficient method for expressing this communication pattern would be to let the processor containing this row broadcast it to all the others. Unfortunately, Java RMI does not support broadcasting, so this cannot be expressed in Java. Instead, we let every machine do an RMI to the processor containing the row for the current iteration. The speedup of the wide-area ASP program for a graph with 768 nodes is given in Figure 3. The program takes 52% longer on four clusters of 10 machines than on a single 40-node cluster. The reason is that, at the beginning of every iteration, most processors have to obtain the row data from a remote cluster. These inter-cluster

RMIs each contain 3200 bytes of data and cost (at least) 13 msec. Still, the program is a factor 2.3 faster on four clusters of 10 machines than on a single 10-node cluster, so it is useful to run ASP on multiple clusters.

4 Related work

We have discussed a Java-centric approach to writing wide-area parallel (metacomputing) applications. Most other metacomputing systems (e.g., Globus [10] and Legion [12]) support a variety of languages. The SuperWeb [1] and Javelin [8] 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) [22].

Most work on metacomputing focuses on how to build the necessary infrastructure [2, 10, 12, 20]. 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 [9, 10, 12, 19]. Coping with such a large non-uniformity in the interconnect complicates application development. In our earlier research, we experimented with optimizing the performance of 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 [19]. Based on this experience, we implemented collective communication operations as defined by the MPI standard, resulting in improved application performance on wide area systems [13]. 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 (see, for example, the JavaGrande Forum at *http://www.javagrande.org/*). Titanium [23] is a Java-based language for high-performance parallel scientific computing. The JavaParty system [18] 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 [17] 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 [15]. Java/DSM [24] implements a JVM on top of a distributed shared memory system. Breg et al. [7] study RMI performance and interoperability. Krishnaswamy et al. [14] improve RMI performance somewhat with caching and by using UDP instead of TCP. 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.

5 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 programming system (Manta) 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 light-weight 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 also encountered several shortcomings of the RMI model, in particular the lack of asynchronous communication and broadcast.

In the near future we will extend the Panda library with support for heterogeneity, to allow RMI communication between different types of processors. In addition, we will do research on programming support that eases the development of efficient wide-area parallel applications.

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 Aske Plaat, Raoul Bhoedjang, Rutger Hofman, and Cees Verstoep for their contributions to this research. We thank Cees Verstoep and John Romein 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, April 1996.

[3] H. 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, February 1998.

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

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

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

[7] 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, February 1998.

[8] 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.

[9] 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.

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

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

[12] A.S. Grimshaw and Wm. A. Wulf. The Legion Vision of a Worldwide Virtual Computer. *Comm. ACM*, 40(1):39–45, January 1997.

[13] 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 *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Atlanta, GA, May 1999.

[14] 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.

[15] 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.

[16] J. Maassen, R. van Nieuwpoort, R. Veldema, H.E. Bal, and A. Plaat. An Efficient Implementation of Java's Remote Method Invocation. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Atlanta, GA, May 1999.

[17] M. W. Macbeth, K. A. McGuigan, and Philip 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.

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

[19] A. Plaat, H. Bal, and R. Hofman. Sensitivity of Parallel Applications to Large Differences in Bandwidth and Latency in Two-Layer Interconnects. In *Fifth International Symposium on High-Performance Computer Architecture*, pages 244–253, Orlando, FL, January 1999. IEEE CS.

[20] 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*, April 1997.

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

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

[23] 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*, February 1998.

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