Remote object detection in cluster-based Java

Gabriel Antoniu, Phil Hatcher

Remote object detection in cluster-based Java

Gabriel Antoniu
LIP, ENS Lyon
46 Allée d'Italie
F-69364 Lyon Cedex 07, France
Contact: Gabriel.Antoniu@ens-lyon.fr

Philip Hatcher
Dept. Computer Science
Univ. New Hampshire
Durham, NH 03824, USA
Contact: Philip.Hatcher@unh.edu

Abstract

Our work combines Java compilation to native code with a run-time library that executes Java threads in a distributed-memory environment with true parallelism. This approach is implemented within the Hyperion system for the distributed execution of compiled Java programs on clusters of PCs. To provide the illusion of a shared memory to Java threads, Hyperion has been built on top of DSM-PM2, a portable implementation platform for multithreaded distributed-shared-memory protocols. We have designed, implemented and experimented with two alternative consistency protocols compliant with the Java Memory Model. The protocols have different mechanisms for access detection: the first one uses explicit locality checks, whereas the second one is based on page faults. We illustrate the effects of the two access-detection techniques with five applications run on two clusters with different interconnection networks: BIP/Myrinet and SISCI/SCI.

1. Introduction

The Java programming language has emerged as an attractive alternative for writing parallel programs and the cluster of computers has emerged as the typical parallel machine today. There are a large number of efforts trying to provide support for a distributed execution of Java programs on clusters, mainly by using Java's remote-method-invocation facility (e.g., [6, 7, 12, 13]) or by grafting an existing message-passing library (e.g., [9, 10]) onto Java, in order to connect multiple Java Virtual Machines.

In contrast, we view a cluster as executing a single Java Virtual Machine (JVM), where the nodes are resources for the distributed execution of Java threads with true concurrency. Distribution must be transparent to the programmer and the implementation of the JVM must support the illusion of a shared memory within the context of the Java Memory Model. Thus, any threaded Java program written for a shared-memory machine would run *with zero changes* in a distributed environment. Our approach is most closely related to efforts to implement Java interpreters on top of a distributed shared memory [4, 8, 15]. However, we favor compiling rather than interpreting, since we are interested in computationally intensive programs that can exploit parallel hardware. We expect the cost of compiling to native code will be recovered many times over in the course of executing such programs. Therefore we focus on combining Java compilation with support for executing Java threads using a distributed-shared-memory runtime system.

Our work is done in the context of the Hyperion environment for the high-performance execution of Java programs. Hyperion was developed at the University of New Hampshire and comprises a Java-bytecode-to-C translator and a run-time library for the distributed execution of Java threads. Hyperion has been built using the PM2 distributed, multithreaded run-time system from the École Normale Supérieure de Lyon [2]. As well as providing lightweight threads and efficient inter-node communication, PM2 provides a configurable, multi-protocol, page-based distributed-shared-memory layer, DSM-PM2 [1], providing full support for implementing various consistency protocols, such as Java consistency. Another important advantage of PM2 is its high portability on several UNIX platforms and on a large variety of network protocols (BIP, SCI, VIA, MPI, PVM, TCP). Thanks to this feature, Java programs compiled by Hyperion can be executed with true parallelism in all these environments.

To provide the illusion of a shared memory to Java threads, we have designed, implemented and experimented with two alternative consistency protocols compliant with the Java Memory Model. Though both protocols follow practically the same algorithmic lines, they differ in the mechanisms used for detecting accesses to remote objects. The first protocol uses explicit locality checks, whereas the second one relies on page faults. In this paper we discuss the two techniques, by illustrating and comparing their behavior on a set of applications.

0-7695-0990-8/01/$10.00 (C) 2001 IEEE

Module	Description
Threads subsystem	Provides support for implementing Java threads. Includes thread creation and synchronization, which are mapped to PM2's corresponding operations.
Communication subsystem	Supports the transmission of messages between the cluster nodes. The interface is based upon message handlers being asynchronously invoked on the receiving end (RPCs).
Memory subsystem	Implements the image of a single memory address space shared by all nodes, while respecting consistency as defined by the Java Memory Model. Utilizes either of two alternative protocols developed using the DSM-PM2 distributed-shared-memory platform.
Load balancer	Handles the distribution of newly created threads to nodes. It currently uses a round-robin thread distribution algorithm.
Java API subsystem	Implements a subset of the native methods present in the API classes (we use Sun's JDK 1.1), thus allowing classes using some native methods to be compiled with Hyperion. Classes in the Java API that do not include native methods can simply be compiled by Hyperion's Java-bytecode-to-C translator.

Table 1. Hyperion's runtime: internal structure

An overview of the Hyperion system is given in Section 2. In Section 3 we present our alternative implementations of Java consistency through two DSM-PM2 consistency protocols with access detection based either on locality checks or on page faults. The behavior of the two protocols is illustrated and discussed in Section 4, based on experiments with five different applications run on two platforms using different interconnection networks: BIP/Myrinet and SISCI/SC. Finally, Section 5 summarizes the conclusions of this study.

2. Compiling threaded Java programs for execution on clusters

2.1. The Hyperion system

Our vision is that programmers will develop Java programs using the workstations on their desks and then submit the programs for production runs to a "high-performance Java execution server", usually a cluster, that appears as a resource on the network. Instead of the conventional Java paradigm of pulling bytecode back to their workstation for execution, programmers will push bytecode to the high-performance server for remote execution. Upon arrival at the server the bytecode is translated for native execution on the processors of the cluster. The code generation process is as follows: user class files are compiled (first by Hyperion's Java-bytecode-to-C compiler and then the generated C code by a C compiler) and then linked with the Hyperion run-time library and with the necessary external libraries.

The Hyperion run-time library is structured as a collection of modules that interact with one another. Their role is concisely described in Table 1.

2.2. Hyperion/PM2 implementation

The current implementation of Hyperion is based on the PM2 distributed multithreaded environment. PM2's programming interface allows threads to be created locally and remotely and to communicate through RPCs (Remote Procedure Calls). PM2 also provides a *thread migration* mechanism that allows threads to be transparently and preemptively moved from one node to another during their execution. Such functionality is typically useful to implement dynamic load balancing policies. Finally, on top of PM2, DSM-PM2 [1] provides a platform for implementing multithreaded DSM consistency protocols for various consistency models, such as *sequential* and *release* consistency. We have used DSM-PM2 to implement Java consistency, as described in [3]. In this paper we compare two alternative protocols for Java consistency based on different access-detection mechanisms for remote objects and we compare their influence on the performance of five applications.

Most Hyperion run-time primitives (mainly in the thread, communication and shared memory subsystems) are implemented by direct mapping onto the corresponding PM2 functions. Thus, thread operations are handled by PM2's thread library (Marcel), an efficient, user-level, POSIX-like thread package featuring thread migration. The Hyperion communication subsystem has been easily implemented through PM2's RPCs, which allow PM2 threads to invoke the remote execution of user-defined services (i.e., functions). On the remote node, PM2 RPC invocations can either be handled by a pre-existing thread or they can involve the creation of a new thread. PM2 utilizes a generic communication package [5] that provides an efficient interface to a wide-variety of high-performance communication libraries, including low-level ones.

loadIntoCache	Load an object into the cache
invalidateCache	Invalidate all entries in the cache
updateMainMemory	Update memory with modifications made to objects in the cache
get	Retrieve a field from an object previously loaded into the cache
put	Modify a field in an object previously loaded into the cache

Table 2. Key DSM primitives

3. Remote object detection in Hyperion

3.1. Implementing the Java Memory Model

Hyperion implements the Java Memory Model [11], which utilizes a variant of release consistency. Java allows threads to keep locally cached copies of objects. Consistency is provided by requiring that local modifications made to cached objects be transmitted to the central memory when a thread exits a monitor. Table 2 provides the key primitives of the Hyperion memory subsystem that are used to provide Java consistency.

The concept of *central memory* is implemented in DSM-PM2 via a *home-based* approach. Each object is assigned a home node, which is in charge of managing the reference copy of the object. Objects (initially stored on their home nodes) are replicated when accessed on other nodes, using loadIntoCache. Note that at most one copy of an object may exist on a node and that this copy is shared by all the threads running on that node. Thus, by associating caches to nodes rather than to threads we avoid wasting memory. Also, objects are implemented on top of pages such that loadIntoCache actually retrieves the whole page on which the object is located, which results in a pre-fetching effect for other objects located on the same page. This is still compliant with Java consistency. The threads are guaranteed to access up-to-date objects since the cache gets invalidated upon entering a monitor.

Thanks to the put access primitives, the modifications can be recorded at the moment when they are carried out, with object-field granularity. All local modifications are sent to the home node of the page by the updateMainMemory primitive, called by the Hyperion run-time on exiting a monitor.

Since DSM-PM2's programming interface is intended both for support of more complex libraries and as a target for compilers, no pre-processing is assumed in the general case and accesses to shared data can be detected using page faults. Alternatively, an application can choose to bypass the fault detection mechanism by controlling the accesses to shared data through calls to get/put primitives that explicitly handle consistency. We used both approaches in our alternative implementations of the Java consistency model.

Note that in both cases, references to objects are real pointers and that the objects are located at the same virtual address on all nodes, according to the *iso-address* approach to memory allocation taken by PM2. Thanks to this scheme, page replication and migration, as well as thread migration, can occur while guaranteeing pointer validity.

3.2. Access detection using in-line checks

Since Hyperion uses specific access primitives to shared data (get and put), we can use explicit checks to detect if an object is present (i.e., has a copy) on the local node, thus bypassing the page-fault mechanism. If the object is present, it is directly accessed, otherwise the page containing the object is brought to the local cache. This scheme is used by the java_ic protocol (ic for in-line check). Its main advantage is that it saves the cost of page faults that would be necessary to detect accesses to non-local pages. As a result, no page protection is necessary on any node, since we assume all accesses to shared data are carried out through the get/put primitives: the shared memory is allocated in READ/WRITE mode on all nodes at initialization time and the access rights remain the same until the end of the application. In conclusion, accesses to remote objects do not involve either page faults, or any call to the mprotect primitive to handle access rights. However, the price to pay is an explicit locality check for *every* access to an object, whether it be local or remote.

3.3. Access detection using page faults

Alternatively, the java_pf protocol uses page faults to detect accesses to non-local objects (pf for page fault). Initially, objects are set to READ/WRITE mode on the home node only and are READ/WRITE-protected on the other nodes. This protection is set on each entry to a monitor, such that any subsequent access to objects on non-home nodes always results in a page fault that starts up the consistency protocol by requesting the page from the home node. Upon arrival, the page access rights are set to READ/WRITE and the faulting thread is thus guaranteed to access up-to-date data (once it has entered the monitor). Compared to the java_ic protocol, accesses to local objects (i.e. objects on their home node or cached) are cheaper, since they no longer involve checks. In contrast, an extra overhead is introduced for the loading of a remote object,

due to the page fault itself and to the calls to the UNIX primitive mprotect.

We can see that choosing between one technique or the other involves a tradeoff which needs to take into account complex factors like the ratio between the number of local accesses to the number of remote accesses (influenced by the data distribution and by the computation-to-communication ratio of the application) and the relative cost of page faults against inline-checks. Thanks to the customizability of DSM-PM2, we could experiment with both techniques. Five applications with different characteristics have been used to analyze this tradeoff.

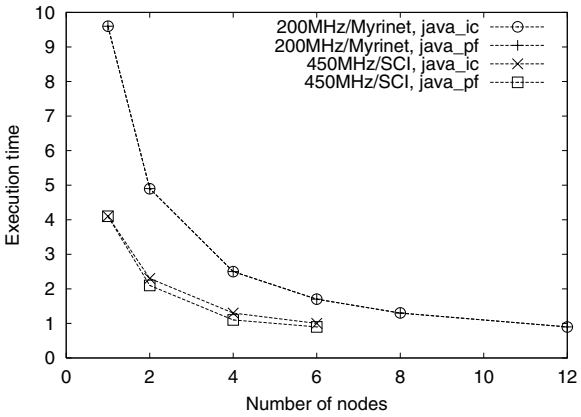

Figure 1. Pi: java_pf vs. java_ic.

4. Performance evaluation

4.1. Benchmark programs

We evaluated the two protocols using five benchmark programs. The Pi program estimates π by calculating a Riemann sum of 50 million values. The Jacobi program computes the temperature distribution on an insulated plate after 100 time steps, using a 1024 by 1024 mesh of cells to discretize the plate. Barnes is a gravitational N-body simulation adapted from the C code distributed with the SPLASH-2 benchmark suite. We used 16K bodies and ran the simulation for 6 timesteps. TSP is a branch-and-bound solution to the Traveling Salesperson Problem, computing the shortest path connecting all cities in a given set. We solved a 17-city problem. ASP is the All-pairs, Shortest Paths program, which computes the shortest path between all pairs of nodes in a graph, using Floyd's algorithm. We used a 2000 node graph. (The ASP and TSP applications are based upon code generously given to us by the Jackal group at the Vrije Universiteit.)

Each program creates one computation thread for each processor in the cluster. Pi is embarrassingly parallel, with threads coordinating only to compute a global sum of the partial sums computed by the threads for their share of the Riemann intervals. In Jacobi each thread owns a block of contiguous rows of the mesh. During every timestep each thread must retrieve a "boundary" row from its "neighbor" thread holding the rows to the "north" and from its "neighbor" thread holding the rows to the "south". The communication pattern in Barnes is irregular as bodies move during the simulation (causing body-body interactions to change) and the program uses a load-balancing algorithm that dynamically assigns bodies to threads for processing. TSP uses a central queue of work to be performed, as well as centrally storing the best solution seen so far. Of course, these "central" data structures are stored on a single node, protected by a Java monitor, and must be fetched by threads executing on other nodes. ASP uses a two-dimensional distance matrix. As in Jacobi, each thread owns a block of contiguous rows of the matrix. During each iteration the "current" row of the matrix must be retrieved by all threads.

4.2. Experimental results

The programs were run on two different clusters. The first cluster consists of twelve 200 MHz Pentium Pro machines, running Linux 2.2, interconnected by a Myrinet network and using the BIP protocol [14]. The second cluster consists of six 450 MHz Pentium II machines running Linux 2.2, interconnected by a SCI network using the SISCI protocol. The cost of a page fault goes from 12 microseconds on the SCI cluster machines to 22 microseconds on the Myrinet cluster machines.

The GNU C compiler (version 2.7.2.3) was used as the "back end" to Hyperion's java2c compiler. The GNU C compiler was invoked using the -O6 optimization flag.

Figures 1-5 compare the two protocol implementations for each benchmark program on both clusters.

4.3. Discussion

The two protocols performed essentially identically on both clusters for the Pi program. This is not surprising as this program makes very little use of objects, computing nearly exclusively with values on each thread's stack. Therefore, java_ic performs very few locality checks and java_pf handles very few page faults, leading to identical performance for the two protocols.

On the Myrinet cluster, java_pf consistently outperforms java_ic for the other applications. The improvements ranged from 38% for Jacobi to 64% for ASP. The most important factor in the amount of the improvement concerns the ratio of the cost of the inline check used by the java_ic protocol to the cost of the rest of the computation

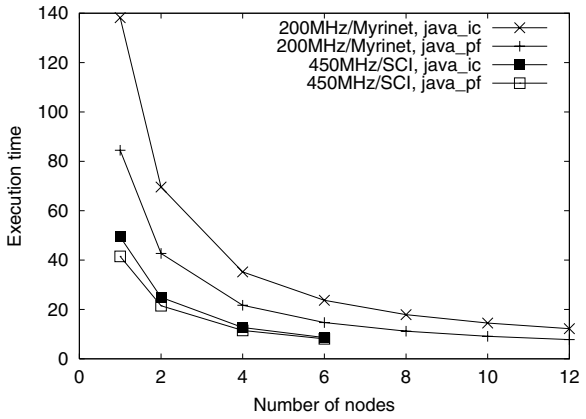

Figure 2. Jacobi: java_pf vs. java_ic.

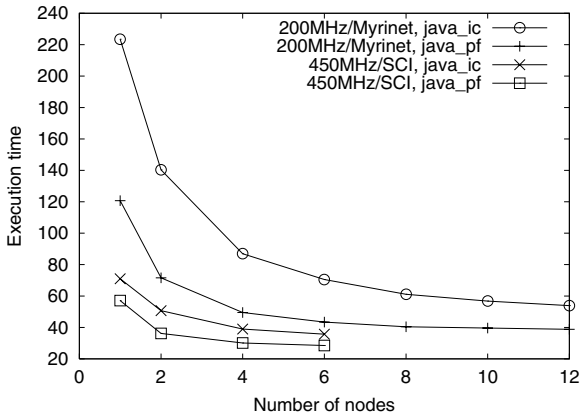

Figure 3. Barnes Hut: java_pf vs. java_ic.

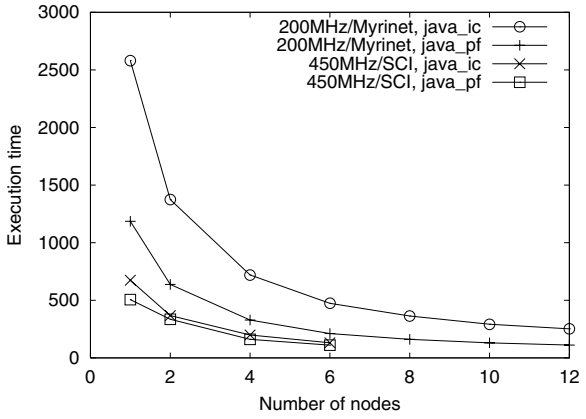

Figure 4. TSP: java_pf vs. java_ic.

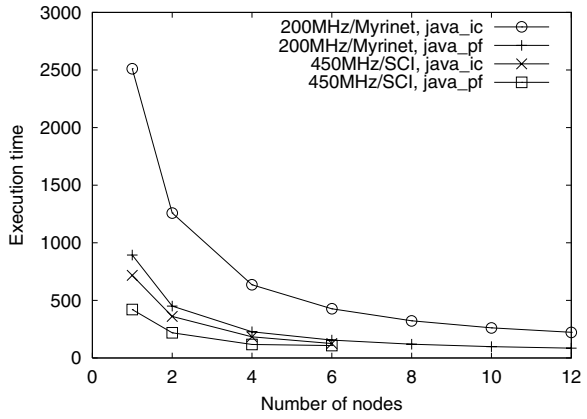

Figure 5. ASP: java_pf vs. java_ic.

performed by the benchmark program. In ASP the innermost loop is only doing an integer add and an integer compare while performing three object-locality checks. Removing these checks obviously has a large impact on the performance. The innermost loop of the Jacobi program, on the other hand, utilizes double-precision floating-point operations, which make the object-locality checks less important.

For Jacobi, TSP and ASP the improvement amount on the Myrinet cluster is relatively constant as the number of nodes is varied. However, for Barnes the improvement decreases from 46% to 28% as the number of nodes increases from 1 to 12. With Barnes, for this problem size on the Myrinet cluster, the execution time using either protocol flattens for the higher number of nodes: communication costs grow steadily and begin to dominate. This means that the number of page faults being handled by java_pf (as well as the number of mprotect calls performed) grows significantly. This eats away at the improvement obtained by removing the locality checks.

The communication costs for Jacobi, TSP and ASP are relatively constant for these problems sizes as the Myrinet cluster size is varied. (For Jacobi the communication costs are constant; for TSP and ASP the communication costs are dwarfed by the computation costs, even at the larger cluster sizes.) Therefore the additional overhead introduced by the page fault handling and page protection actions in java_pf is not significant.

On the SCI cluster, java_pf also consistently outperforms java_ic for the Jacobi, Barnes, TSP and ASP applications. However, the improvement is less than on the Myrinet cluster: an average improvement across all applications and cluster sizes of 21%. This reflects the faster speed of the processors used in the SCI cluster, which makes the removal of the in-line checks relatively less important.

These results indicate that the page-fault protocol is su-

perior for all applications and clusters studied. For parallel programs that exhibit good speedup as the number of cluster nodes is increased, the additional overheads incurred by the page-fault handling will not be significant since the communication costs of such programs are not dominant. In this case the amount of improvement observed on a single-node execution by removing the checks should also be observed on runs with a larger number of cluster nodes. The amount of improvement will vary depending on how expensive the locality checks are with respect to the overall computation. On the other hand, note that we used only one application thread per node. We also plan to study the effects of using more application threads per node, thus enabling computation/communication overlap.

5. Conclusion

We have compared two alternative remote-object-detection techniques embedded within two different consistency protocols implementing Java consistency. This study has been performed within the context of a DSM-based Java compilation system for clusters, Hyperion. Both protocols allow threaded Java programs written for shared memory to run in a distributed environment with no changes. To detect accesses to remote objects, the first protocol (java_ic) uses explicit locality checks, whereas the second protocol (java_pf) relies on page faults. We have used five different applications on two different platforms (BIP/Myrinet and SISCI/SCI) to compare the behavior of our protocols. The experimental results clearly demonstrate that the protocol based on page faults is superior. Even if remote object retrieval is more expensive for this protocol (due to fault handling and to page protection overhead), the absence of any check in local accesses makes it much more efficient than java_ic, which introduces a check for *every* access. This behavior can be explained by the intensive use of objects combined with a "good" object distribution: local objects are intensively used, remote accesses (generating faults for java_pf) are not very frequent.

We stress that both DSM-PM2 protocols for Java consistency were co-designed with Hyperion's memory module and this approach enabled us to make aggressive optimizations using information from the upper layers. This illustrates how the use of a customizable, portable platform for implementing multithreaded consistency protocols such as DSM-PM2 can allow for easy experimentation with different design techniques and lead to efficient choices. We plan to use this feature to experiment with other mechanisms to implement Java consistency, including thread migration.

References

[1] G. Antoniu and L. Bougé. DSM-PM2: A portable implementation platform for multithreaded DSM consistency protocols. In *Proc. 6th International Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS '01)*, San Francisco, Apr. 2001. Held in conjunction with IPDPS 2001. IEEE TCPP. To appear.

[2] G. Antoniu, L. Bougé, P. Hatcher, M. MacBeth, K. McGuigan, and R. Namyst. Compiling multithreaded Java bytecode for distributed execution. In *Euro-Par 2000: Parallel Processing*, volume 1900 of *Lect. Notes in Comp. Science*, pages 1039–1052, Munchen, Germany, Aug. 2000. Distinguished paper.

[3] G. Antoniu, L. Bougé, P. Hatcher, M. MacBeth, K. McGuigan, and R. Namyst. Implementing Java consistency using a generic, multithreaded DSM runtime system. In *Proc. Intl Workshop on Java for Parallel and Distributed Computing*, volume 1800 of *Lect. Notes in Comp. Science*, pages 560–567, Cancun, Mexico, May 2000. Held in conjunction with IPDPS 2000.

[4] Y. Aridor, M. Factor, and A. Teperman. cJVM: A single system image of a JVM on a cluster. In *Proceedings of the International Conference on Parallel Processing*, Fukushima, Japan, Sept. 1999.

[5] L. Bougé, J.-F. Méhaut, and R. Namyst. Efficient communications in multithreaded runtime systems. In *Proc. 3rd Workshop on Runtime Systems for Parallel Programming (RTSPP '99)*, volume 1586 of *Lect. Notes in Comp. Science*, pages 468–182, San Juan, Puerto Rico, Apr. 1999. Held in conjunction with IPPS/SPDP 1999.

[6] F. Breg, S. Diwan, J. Villacis, J. Balasubramanian, E. Akman, and D. Gannon. Java RMI performance and object model interoperability: Experiments with Java/HPC++. In *Proceedings of the ACM 1998 Workshop on Java for High-Performance Network Computing*, pages 91–100, Palo Alto, California, February 1998.

[7] D. Caromel, W. Klauser, and J. Vayssiere. Towards seamless computing and metacomputing in Java. *Concurrency: Practice and Experience*, 10:1125–1242, 1998.

[8] X. Chen and V. Allan. MultiJav: A distributed shared memory system based on multiple Java virtual machines. In *Proceedings of the Conference on Parallel and Distributed Processing Techniques and Applications*, Las Vegas, Nevada, June 1998.

[9] A. Ferrari. JPVM: Network parallel computing in Java. In *Proceedings of the ACM 1998 Workshop on Java for High-Performance Network Computing*, pages 245–249, Palo Alto, California, 1998.

[10] V. Getov, S. Flynn-Hummell, and S. Mintchev. High-performance parallel programming in Java: Exploiting native libraries. In *Proceedings of the ACM 1998 Workshop on Java for High-Performance Network Computing*, pages 45–54, Palo Alto, California, February 1998.

[11] J. Gosling, W. Joy, and G. Steele Jr. *The Java Language Specification*. Addison-Wesley, Reading, Massachusetts, 1996.

[12] J. Maassen, T. Kielmann, and H. Bal. Efficient replicated method invocation in Java. In *Proceedings of the ACM 2000 Java Grande Conference*, pages 88–96, San Francisco, California, June 2000.

[13] M. Philippsen and M. Zenger. JavaParty — transparent remote objects in Java. *Concurrency: Practice and Experience*, 9(11):1125–1242, November 1997.

[14] L. Prylli and B. Tourancheau. BIP: A new protocol designed for high performance networking on Myrinet. In *Proceedings of First Workshop on Personal Computer Based Networks Of Workstations*, volume 1388 of *Lect. Notes in Comp. Science*, pages 472–485. Springer-Verlag, Apr. 1998.

[15] W. Yu and A. Cox. Java/DSM: A platform for heterogeneous computing. In *Proceedings of the Workshop on Java for High-Performance Scientific and Engineering Computing*, Las Vegas, Nevada, June 1997.