Enabling Java for High-Performance Computing: Exploiting Distributed Shared Memory and Remote Method Invocation

Thilo Kielmann[*] Philip Hatcher Luc Bougé Henri E. Bal

Java has become increasingly popular as a general-purpose programming language. Current Java implementations mainly focus on portability and interoperability, which is required for Internet-centric client/server computing. Key to Java's success is its intermediate "bytecode" representation that can be exchanged and executed by *Java Virtual Machines* (JVMs) on almost any computing platform. Along with Java's widespread use, the need for a more efficient execution mode has become apparent. For sequential execution, *Just-in-Time* (JIT) compilers improve application performance [4]. However, high-performance computing typically requires multiple-processor systems, so efficient interprocessor communication is needed in addition to efficient sequential execution.

Being an object-oriented language, Java uses *method invocation* as its main concept of communication. Inside a single JVM, concurrent threads of control can communicate by synchronized method invocations. On a multiprocessor system with shared memory (SMP), this approach allows for some limited form of true parallelism by mapping threads to different physical processors. For distributed-memory systems, Java offers the concept of a *Remote Method Invocation* (RMI). Here, the method invocation, along with its parameters and results, is transferred across a network to and from the serving object on a remote JVM.

With these concepts for concurrency and distributed-memory communication, Java provides a hitherto unique opportunity for a widely accepted general-purpose language with a large existing code and programmer base that can also suit the needs of parallel (high-performance) computing. Unfortunately, Java has not yet been widely perceived as such, due to the inefficiency of current implementations. In this treatise, we provide evidence of the usefulness of Java for parallel computing by describing efficient implementation techniques. We show that the combination of powerful compilers and efficient runtime systems leads to Java execution environments that can successfully exploit the computational power of distributed-memory parallel computers, scaling to system sizes unreachable for pure shared-memory approaches.

A major advantage of Java is that it provides communication mechanisms inside the language environment, whereas other languages (e.g., Fortran or C++) require external mechanisms (libraries) like message passing. In fact, bindings of the *Message Passing Interface* standard (MPI) for Java already exist [5]. However, the MPI message-passing style

[*]contact author at `Thilo.Kielmann@acm.org`.

of communication is difficult to integrate cleanly with Java's object-oriented model, especially as MPI assumes a *Single-Program, Multiple-Data* (SPMD) programming model that is quite different from Java's multithreading model. In this treatise we try to show that, with efficient compilers and runtime systems, pure Java is a platform well suited for parallel computing. We pursue two approaches for achieving this goal.

The first approach allows truly parallel execution of multi-threaded Java programs on distributed memory platforms. This idea is implemented in the Hyperion system, which compiles multithreaded bytecode for execution on a "distributed virtual machine". Hyperion provides efficient inter-node communication and a distributed-shared-memory layer through which threads on different nodes share objects. This is the purest approach to Java-based parallel computing on distributed-memory systems: Hyperion completely hides the distributed-memory environment from the application programmer, and allows any object to be accessed from any machine.

The second approach we present provides the programmer with the explicit notion of shared objects. Here, the programmer indicates which objects will be shared among multiple threads. Communication between threads is reduced to method invocations on such shared objects. This approach is implemented in the Manta system. Manta statically compiles Java source code to executable programs; its runtime system provides highly efficient RMI as well as a similar mechanism called *Replicated Method Invocation* (RepMI), allowing for more efficient use of object locality.

For both Hyperion and Manta, we present the basic implementation techniques that lead to efficient parallel execution of Java programs on distributed-memory platforms. We provide performance data for the respective communication operations and discuss the suitability of the approaches to parallel programming. We compare both the promise and the limitations of the two approaches for Java-centric parallel computing.

Hyperion: transparent distributed multithreading

Hyperion allows a Java programmer to view a cluster of processors as executing a *single* JVM [1]. In Java, concurrency is exposed to the user through *threads* sharing a common address space. The standard library provides facilities to start a thread, suspend or kill it, switch control between threads, etc., and the *Java memory model* specifies how threads may interact with the common memory (see sidebar.) Thus it is possible to map a multi-threaded Java program onto a cluster directly. Faster execution is obtained by mapping the original Java threads onto the native threads available in the cluster. These threads are spread across the processing nodes to provide actual concurrent execution and load balancing. The Java memory model is implemented by a *distributed shared memory* (DSM) substrate, so that the original semantic model of the language is kept unchanged.

For efficient execution, Hyperion compiles Java bytecode into optimized native code. This is done in a two-step process. Java bytecode is first translated to C, and then a C compiler is used to generate native code for the processors of the cluster. Using a C compiler for generating native code provides the benefits of platform-specific compiler optimizations while keeping the system itself platform-independent.

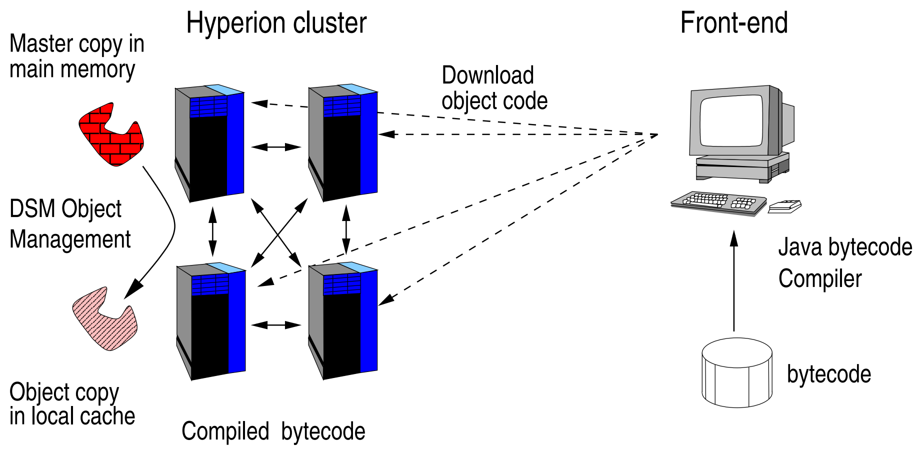

Figure 1: Management of distributed-object memory in Hyperion.

Portability has been a major objective in the design of Hyperion. Therefore, the runtime system has been built on top of a portable environment called *DSM-PM2*, which extends the multithreaded library PM2 [9] with a DSM facility. DSM-PM2 provides lightweight multithreading, high-performance communication, and a page-based DSM. It is portable across a wide spectrum of high-performance networks, such as the *Scalable Coherent Interface* (SCI), Myrinet, and Gigabit-Ethernet, and can be used with most common communication interfaces, such as the standard TCP protocol, MPI and the *Virtual Interface Architecture* (VIA).

The central aspect of Hyperion's design is the management of the distributed-object memory (see Figure 1). Hyperion's programming model must provide the illusion of a uniformly accessible, shared-object memory, which is independent of the physical object locations. According to the original specification of the Java memory model, each thread in Hyperion is conceptually equipped with a *local cache*, interacting with a common *main memory*. Caching greatly improves performance if the application exhibits temporal locality, accessing a cached object multiple times before the cache is invalidated.

Hyperion is not unique in its goal of providing a fully transparent cluster implementation of Java. Java/DSM [12], cJVM [2] and JESSICA [6] are examples of similar systems. However, they are based on interpreted bytecode rather than native compilation. While the above systems differ in their approaches to implementing the Java Memory Model, and in the type of their target applications, collectively they demonstrate the potential of using Java to efficiently utilize clusters. Systems such as Hyperion also draw heavily on the extensive body of DSM-related literature.

Hyperion is a research prototype and currently only supports parts of the standard Java libraries, limiting its ability to inter-operate with other JVMs. Also, Hyperion currently does not support dynamic class loading, which would require implementing the dynamic compiling of bytecode and the dynamic linking of native code.

Table 1: Completion times (in microseconds) of elementary DSM operations on a Pentium Pro/Myrinet cluster.

	local	cached	remote
read	0.02	0.02	370
write	0.04	0.50	480
sync	2.70	—	180

Performance

The net result of Hyperion's implementation techniques is that it provides efficient execution of unmodified Java programs on a wide range of distributed clusters. We believe this flexibility is a major incentive for Java users in search of high performance. Table 1 presents timings of local, cached and remote elementary DSM operations, measured with Hyperion on a cluster of Pentium Pros running at 200 MHz, communicating over Myrinet.

The first two lines display the time in microseconds to access local, cached, and remote objects on this platform. Remote access times include the costs of transferring the page containing the object. The page size is 4096 bytes. In detail, a remote read operation includes: detecting the absence of the object and transmitting the request (114 μs, 30 %), transferring the page across the network (134 μs, 37 %), additional Hyperion-level processing (122 μs, 33 %). Writing to a cached copy of an object involves recording the modifications for later consistency updates, adding 0.48 μs to reading. Writing to a remote object is more expensive than reading it, because a remote write must transmit the modification back to the home location.

The last line displays the time to perform remote and local synchronization. This is the time to enter and exit a Java monitor (see sidebar on multithreading.) In the remote case, the lock being accessed by the monitor is on a different node.

Manta: efficiently shared objects

Manta uses a different philosophy than Hyperion. Instead of providing a shared-memory programming model, Manta requires the programmer to store shared data in remote objects, which can be accessed using Java's RMI (see the respective sidebar). Manta's programming model is the same as that of standard RMI, plus a simple extension that allows the programmer to improve locality by indicating which objects should be replicated. Manta implements this programming model in a highly efficient way, using a completely new implementation of Java and RMI [8].

Efficient RMI

Manta uses a native off-line Java compiler that statically compiles Java source code into executable binary code. Using a static compiler allows aggressive, time-consuming opti-

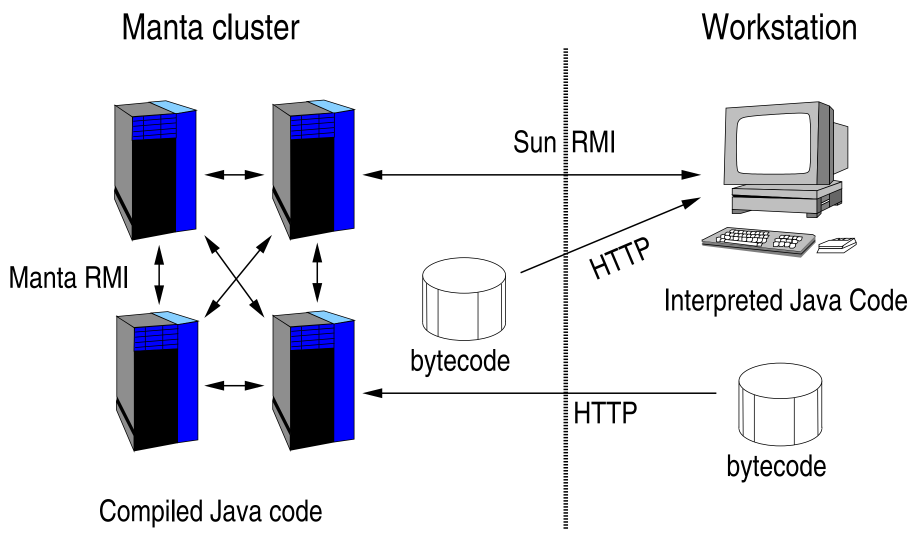

Figure 2: Interoperability of Manta and Sun RMI.

mizations. Manta's fast RMI implementation consists of three components:

- A new light-weight RMI protocol. This protocol is completely implemented in C, avoiding the layering overhead of other RMI systems that invoke low-level C routines from Java code via the slow *Java Native Interface* (JNI). The protocol minimizes the overhead of thread switching, buffer management, data format conversions (byte swapping), and copying.
- Object serialization. The Manta compiler generates specialized serialization routines for serializable argument classes, avoiding the overhead for runtime type inspection that is typical of most other Java systems.
- Efficient communication software. Manta is implemented on top of the Panda communication library [3], which provides message passing, *Remote Procedure Call* (RPC), and broadcasting. On Myrinet, Panda uses a highly-optimized low-level communication substrate. On Ethernet, Panda uses the standard UDP protocol.

The RMI implementation described so far is compatible with the Java language specification, but uses a different communication protocol. However, Manta uses additional mechanisms to interoperate with other Java Virtual Machines [8], as illustrated in Figure 2. A parallel Java program compiled with Manta's native compiler runs on a cluster. The processes of this application use Manta's fast RMI protocol to communicate with each other. They can also communicate with applications that run on standard JVMs using the standard RMI protocol. They can even exchange bytecode with these applications, which is required for polymorphic RMIs [11]. For this purpose, the Manta compiler also generates bytecode for Java programs (which can be sent to remote JVMs), and the Manta runtime system contains a compiler to process incoming bytecode from a JVM. The net result is that Manta provides efficient sequential code, fast communication, interoperability with JVMs, and polymorphic RMIs. Manta's RMI combines the efficiency of a C-like RPC and the flexibility of Java RMI. The JavaParty project [10] implemented similar optimizations to Java RMI, but without interoperability to standard JVMs. Because JavaParty's RMI is implemented in pure Java, it is also less efficient than Manta's RMI.

Replicated objects

Even with all the optimizations performed by Manta, method invocations on shared objects are much slower than *sequential* method invocation (i.e., an invocation on a normal Java object that is not declared to be remote). Even within the same address space, accessing a *remote* object is costly. Manta addresses this problem with the concept of replicated method invocation (RepMI) [7]. With RepMI, shared objects are replicated across the processes of a parallel application. The advantage of RepMI is that methods which do not modify a replicated object (read-only methods) can be performed on the local copy. Such methods are recognized by the Manta compiler and are executed without any communication, resulting in completion times close to sequential method invocation. Manta also provides a mechanism to replicate collections of objects, such as trees or graphs.

To obtain high performance, RepMI implements methods that do modify a replicated object (write methods) using an update protocol with function shipping, which is the same approach as successfully used in the Orca system [3]. This protocol updates all copies of a replicated object by broadcasting the write-operation and performing the operation on all replicas. The broadcast protocol is provided by the Panda library [3]; it uses totally ordered broadcasting, so that all replicas are updated consistently.

Performance

Table 2 presents timings of local, remote, and replicated method invocations, measured with Manta on a Myrinet cluster with 200 MHz Pentium Pros. The remote write method costs 41 μs. Calling a remote read method requires additional serialization of the result data and costs 42 μs. In comparison, a parameter-less invocation of the underlying Panda RPC protocol takes 31 μs.

Example Applications

We have evaluated our approaches with two small example applications. The performance of the systems has been measured on two clusters with identical processors (200 MHz Pentium Pros) and networks (Myrinet). We present application runtimes, compared to sequential execution with a state-of-the-art Just-in-Time compiler, the IBM JIT 1.3.0.

The first application is the *All-pairs Shortest Paths* (ASP) program, computing 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.

Table 2: Completion times of read and write operations (in microseconds) on a Pentium Pro/Myrinet cluster.

		completion time	
	CPUs	*void write(int i)*	*int read()*
sequential		0.10	0.08
RMI, local		14.96	15.20
RMI, remote		40.63	41.83
replicated	1	21.19	0.33
replicated	2	55.48	0.33
replicated	4	62.61	0.33
replicated	8	70.36	0.33
replicated	16	77.18	0.33
replicated	32	113.20	0.33
replicated	64	118.80	0.33

For the shared-memory version of ASP, used by Hyperion, a single thread is allocated on each processor. Each thread owns a contiguous block of rows of the graph's shared distance matrix. On each iteration each thread fetches the necessary row, updates its own rows, and then synchronizes to wait for all other threads to finish the iteration. Figure 3 shows that the program performs well on small clusters. (The cluster available to Hyperion has only eight nodes.) However, having all threads request the current row separately is likely to limit the scalability. This situation might best be addressed by extending Hyperion's programmer interface to include methods for collective communication among the threads of a thread group.

In the RMI version, each row of the distance matrix simply implements the interface `java.rmi.Remote`, making it accessible for threads on remote nodes. The processor owning the row for the next iteration stores it into its remotely accessible object. Because each machine has to fetch each row for itself, each row has to be sent across the network multiple times (just as with Hyperion), causing high overhead on the machine that owns the row. The replicated ASP implementation uses replicated objects for the rows. Whenever a processor writes a row into its object, the new row is forwarded to all machines. Each processor can then read this row locally. Figure 3 shows that the RMI version performs well up to 16 nodes. On more nodes, the overhead for sending the rows becomes prohibitive. With 64 nodes, the RMI version completes after 38 seconds while the RepMI variant needs only 18 seconds. This difference is due to the efficient broadcast of Manta's runtime system.

The second example application is the *Traveling Salesperson Problem* (TSP), computing the shortest path along all cities in a given set. We use a branch-and-bound algorithm pruning large parts 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 nodes dynamically.

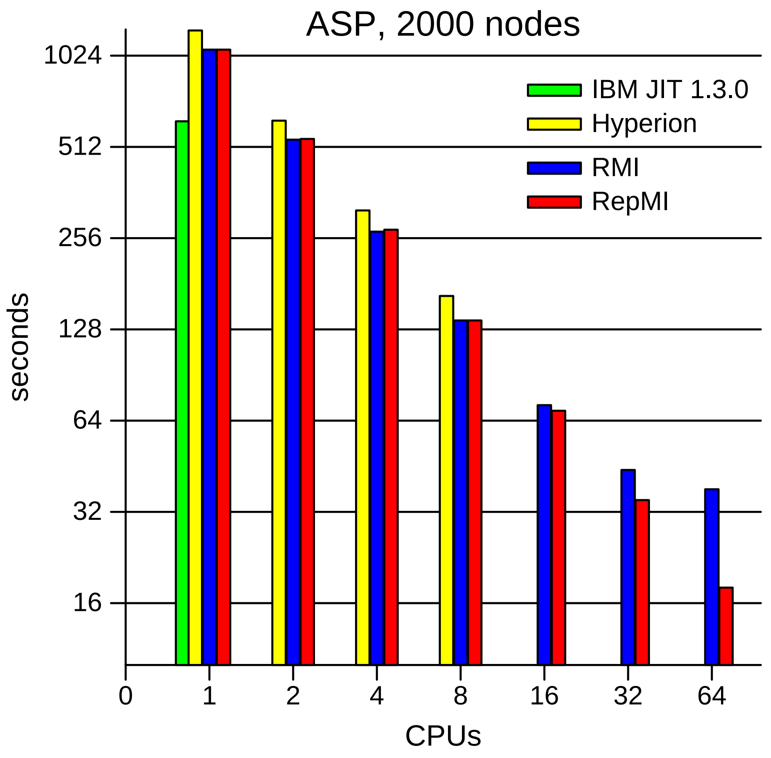

Figure 3: ASP execution times with Hyperion, RMI, and RepMI. (The cluster available to Hyperion has only eight nodes.)

The TSP program keeps track of the best solution found so far. Each node needs an up-to-date copy of this solution to prevent it from doing unnecessary work, causing it to read the value frequently. In contrast, updates happen only infrequently.

The Hyperion shared-memory version again uses a single thread per node. The object containing the hitherto best solution is protected by a monitor (see sidebar on multithreading.) The program scales well on small clusters because of Hyperion's lightweight implementation of its DSM primitives and the application's favorable ratio of local computation to remote data access.

In an RMI version, the overhead of frequently reading a single, remote *Minimum* object would result in poor performance. Instead, a manually optimized version has to be used in which the threads read the minimum value from a local variable. When a thread finds a better minimum value, it invokes an updating RMI on all peer threads which have to be remote objects for this purpose. In contrast, the replicated version of TSP is simple and intuitive. Here, the global *Minimum* object implements the replication interface. All changes to this object are automatically forwarded. Each node can locally invoke the read method of the object, only slightly more slowly than reading a variable directly. While being as simple as the Hyperion version, on 64 nodes, the replicated version completes in 31 seconds, almost as fast as the very complex, manually optimized RMI version which needs 28 seconds.

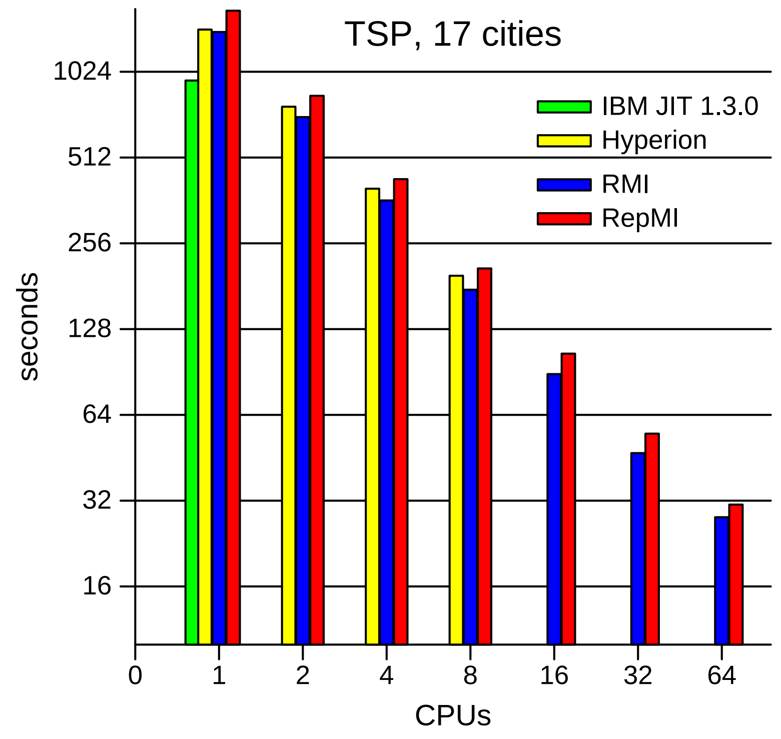

Figure 4: TSP execution times with Hyperion, RMI, and RepMI. (The cluster available to Hyperion has only eight nodes.)

Conclusions

With efficient implementations like the ones provided by Hyperion and Manta, Java provides an unprecedented opportunity: a widely accepted general-purpose language can suit the needs of high-performance computing. Furthermore, Java provides a unique way of rapidly prototyping parallel applications: Starting with a single JVM, parallel applications can be developed based on multithreading. On a small scale, a JVM enables truly parallel thread execution on a multiprocessor machine with shared memory (SMP). For utilizing larger numbers of CPUs, Hyperion-like systems provide transparent execution of multithreaded programs on distributed systems.

Allowing Hyperion programmers to view the cluster as a *black box* is a two-edged sword, however. On the one hand, it allows them to abstract from the internal details of the cluster, that is, individual nodes with private memories. On the other hand, efficient parallel execution can only be provided if each thread predominantly references data that is local, or locally cached. If this is not the case, the communication costs of accessing remote data severely limit the performance improvement obtainable by spreading the threads across the multiple nodes of a cluster. Such multithreaded Java programs can then be converted into programs that make explicit use of shared objects or replicated objects. This conversion requires the programmer to determine which objects will be shared or replicated, and to adapt the program to use RMI to access such shared objects.

Given a high-performance implementation of RMI as with Manta, such programs can obtain high efficiencies even on large-scale, distributed-memory machines.

Acknowledgments

Manta was designed and implemented in a project led by Henri Bal, in cooperation with Thilo Kielmann, Jason Maassen, Rob van Nieuwpoort, Ronald Veldema, Rutger Hofman, Ceriel Jacobs, and Aske Plaat. The lower-level Panda and Myrinet communication software was developed by Raoul Bhoedjang, Tim Rühl, Rutger Hofman, Ceriel Jacobs, and Kees Verstoep. The work on Manta is supported in part by a USF grant from the Vrije Universiteit.

Hyperion was designed and implemented in a project led by Phil Hatcher, in collaboration with Mark MacBeth and Keith McGuigan. The Hyperion-PM2 interface was built in a project led by Luc Bougé, in collaboration with Gabriel Antoniu, who is also the primary author of the DSM support within the PM2 system. Raymond Namyst, Jean-François Méhaut, Olivier Aumage, Vincent Danjean, and other members of the PM2 team have also been essential to the success of the Hyperion project. The Hyperion-PM2 collaboration was supported by funding from NSF and INRIA via the USA-France Cooperative Research program.

References

[1] G. Antoniu, L. Bougé, P. Hatcher, M. MacBeth, K. McGuigan, and R. Namyst. The Hyperion system: Compiling multithreaded Java bytecode for distributed execution. *Parallel Computing*, 2001. To appear.

[2] Y. Aridor, M. Factor, A. Teperman, T. Eilam, and A. Schuster. Transparently obtaining scalability for Java applications on a cluster. *Journal of Parallel and Distributed Computing*, 60(10):1159–1193, Oct. 2000.

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

[4] M. Burke, J.-D. Choi, S. Fink, D. Grove, M.Hind, V. Sarkar, M. Serrano, V. C. Sreedhar, H. Srinivasan, and J. Whaley. The Jalapeño Dynamic Optimizing Compiler for Java. In *ACM 1999 Java Grande Conference*, pages 129–141, San Francisco, CA, June 1999.

[5] B. Carpenter, V. Getov, G. Judd, A. Skjellum, and G. Fox. MPJ: MPI-like Message Passing for Java. *Concurrency: Practice and Experience*, 12(11):1019–1038, 2000.

[6] M. J. M. Ma, C.-L. Wang, and F. C. M. Lau. JESSICA: Java-enabled single-system-image computing architecture. *Journal of Parallel and Distributed Computing*, 60(10):1194–1222, Oct. 2000.

[7] J. Maassen, T. Kielmann, and H. E. Bal. Efficient Replicated Method Invocation in Java. In *ACM 2000 Java Grande Conference*, pages 88–96, San Francisco, CA, June 2000.

[8] 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.

[9] R. Namyst and J.-F. Méhaut. PM2: Parallel multithreaded machine. A computing environment for distributed architectures. In *Parallel Computing (ParCo '95)*, pages 279–285. Elsevier Science Publishers, Sept. 1995.

[10] M. Philippsen, B. Haumacher, and C. Nester. More efficient serialization and RMI for Java. *Concurrency: Practice and Experience*, 12(7):495–518, 2000.

[11] J. Waldo. Remote Procedure Calls and Java Remote Method Invocation. *IEEE Concurrency*, 6(3):5–7, July–September 1998.

[12] W. Yu and A. Cox. Java/DSM: A platform for heterogeneous computing. *Concurrency: Practice and Experience*, 9(11):1213–1224, Nov. 1997.

Sidebar: Java multithreading

Threads in Java are represented as objects. The class `java.lang.Thread` contains methods for initializing, running, suspending, querying and destroying threads. All threads share the same central memory, so all objects are accessible by every thread. Critical sections of code can be protected by monitors. Monitors in Java are available through the use of the keyword `synchronized` and utilize the lock that is associated with every object. For example, a synchronized method first locks the instance of the object it was called on, then the method body is executed, and finally the lock is released. Figure 5 displays the synchronized methods used to access a centralized job queue. Threads may also use methods from `java.lang.Object` to `wait` for an event and to `notify` other threads that an event has occurred.

The Java memory model allows threads to keep locally cached copies of objects. Consistency is provided by requiring that a thread's object cache be flushed upon entry to a monitor and that local modifications made to cached objects be transmitted to the central memory when a thread exits a monitor. This "relaxed" consistency model allows concurrent reading and writing of cached copies of objects by all threads. If shared data is concurrently accessed by multiple threads without proper synchronization, nondeterministic program behavior is possible.

This possibility is often not well understood by current Java programmers who have only experienced the language in the context of a single-processor environment. Consequently, the Java memory model is now receiving considerable attention and the future of the current specification is unclear. William Pugh maintains a Web page as a starting point for discussions concerning the Java Memory Model and its evolution at `http://www.cs.umd.edu/~pugh/java/memoryModel/`.

```
class JobQueue {

  Job[] jobArray;
  int size, first, last, count;

  JobQueue(int size) {
    this.size = size;
    jobArray = new Job[size];
    first = last = count = 0;
  }

  synchronized void addJob(Job j) { /* details omitted for brevity */ }

  synchronized Job getJob() {
    if (count <= 0) return null;

    Job firstJob = jobArray[first];

    first++;
    if (first >= size) first = 0;
    count--;

    return firstJob;
  }
}
```

Figure 5: Java monitor protecting multithreaded access to a shared queue.

Sidebar: Remote Method Invocation (RMI)

Java's Remote Method Invocation (RMI) model allows a client machine to invoke a method on a remote server machine using syntax and semantics that are similar, but not identical, to that of a sequential method invocation. A remote server object, also called *remote object*, is an instance of a class implementing (an extension of) the special `Remote` interface. The server has to register its remote interface with a centralized *registry* and the client looks up the object in this registry. This latter call generates a stub for the remote object on the client machine, and invocations on this stub are automatically forwarded to the server. The programmer also has to provide exception handlers for communication failures, so RMI is not transparent. Moreover, the parameters and return values of a remote call are passed by value in an RMI, i.e., they are copied. The exception is when remote objects are used as parameters: these are passed by reference. For non-remote calls, all objects are passed by reference. Any object of a class implementing the `Serializable` interface can be passed as a parameter of an RMI. The object is automatically serialized (encoded in a network data format), transmitted, and deserialized at the server. The reply is handled similarly.

```
interface PrintServer extends java.rmi.Remote {
   public void print(Serializable obj) throws RemoteException;
}

class ServerObject extends java.rmi.server.UnicastRemoteObject implements PrintServer {
   public ServerObject() throws RemoteException { /* constructor */ }
   public void print(java.io.Serializable obj) throws RemoteException {
      System.out.println("ServerObject received : " + obj.toString());
   }
}

class ClientObject {
   public static void main(String arg[]) {
      String message = "hello";
      try{
         PrintServer server = (PrintServer)Naming.lookup( ... );
         server.print(message);
      }
      catch (Exception e){
         System.out.println("ClientObject exception: " + e.getMessage());
      }
   }
}
```

Figure 6: RMI example.

In Figure 6, the class `ServerObject` implements the interface `PrintServer`. An instance of class `ClientObject` can look up an implementation of `PrintServer` from the registry and invoke the `print` method with any serializable object.