CONCURRENCY—PRACTICE AND EXPERIENCE
Concurrency: Pract. Exper. 2001; **00**:1–6 *Prepared using* cpeauth.cls *[Version: 2001/03/05 v2.01]*

Java for High-Performance Network-Based Computing: A Survey

M. Lobosco[1,*,†],C. Amorim[1,†], and O. Loques[2,‡]

[1] *COPPE - Engenharia de Sistemas, Universidade Federal do Rio de Janeiro*
[2] *Instituto de Computação - Universidade Federal Fluminense*

SUMMARY

There has been an increasing research interest in extending the use of Java towards high-performance demanding applications such as scalable web servers, distributed multimedia applications, and large-scale scientific applications. However, extending Java to a multicomputer environment and improving the low performance of current Java implementations pose great challenges to both the systems developer and application designer. In this survey, we describe and classify fourteen relevant proposals and environments that tackle Java's performance bottlenecks in order to make the language an effective option for high-performance network-based computing. We further survey significant performance issues while exposing the potential benefits and limitations of current solutions in such a way that a framework for future research efforts can be established. Most of the proposed solutions can be classified according to some combination of three basic parameters: the model adopted for inter-process communication, language extensions, and the implementation strategy. In addition, where appropriate to each individual proposal, we examine other relevant issues, such as interoperability, portability, and garbage collection.

KEY WORDS: Java, parallel JVM implementation, high-performance computing, network-based computing

1. INTRODUCTION

Java [4] is an object-oriented programming language, developed by Sun Microsystems, which incorporates features such as multithreading and primitives for concurrent programming. One

*Correspondence to: PESC/COPPE/UFRJ - P. O. Box 68.511 - CEP 21.945-970 - Rio de Janeiro - Brazil
†E-mail: {lobosco, amorim}@cos.ufrj.br
‡E-mail: loques@ic.uff.br
Contract/grant sponsor: CAPES, CNPq, FAPERJ and FINEP

of its main objectives is to allow the portability of programs among different hardware and operating system platforms. This objective is portrayed by the well-known slogan "Write once, run everywhere". The approach taken to reach this goal was the adoption of a standardized supporting platform called the Java Virtual Machine (JVM). The Java compiler generates a platform independent pseudo-code, called *bytecode*, which can then be executed in any computational environment (hardware & operating system) that supports the Java bytecode interpreter included in the standard JVM. The price paid for the portability, achieved through interpretation, as one might expect, is performance. Several attempts intending to improve Java execution performance have been made, such as the addition of just-in-time compilation support and other optimizations techniques to Java execution environments [43]. Recent results showed that optimized Java code performed comparably to Fortran for some numerically-intensive regular computations [38]. However, these improvements were not enough to ensure that Java performed as well as C. Nevertheless, numerous systems for high-performance network computing developed to support Java applications have been proposed in recent years. The applications of these systems tend to be those of a large-scale computational nature, potentially requiring any combination of computers, networks, I/O, and memory, as defined by the Java Grande Forum [27]. Examples of such applications include data mining, satellite image processing, scalable web servers, and fundamental physics. At first glance, the choice of Java seems paradoxical, since it is an interpreted language. This single feature, however, has not been enough to dampen the great interest in its use in the development of high-performance computing environments.

Then, why should we use Java for High-Performance Computing? Besides the portability and interoperability achieved by a standard supporting environment, other features of the language such as its object-oriented programming model, simplicity, robustness, multithreading support, and automatic memory management have proved attractive enough for the development of software projects, especially those intended for large and complex systems. Also, the language portability has been decisive for its choice in projects that consider the use of idle computers, connected to the Internet, to solve large computational problems [7, 10, 21]. In addition, the growing popularity of the language helps to explain its use in the high-performance computing area.

In this survey we describe and classify some Java-based projects aimed directly or indirectly at supporting the development of high-performance networked computing applications. For classification purposes, some parameters, including the inter-process communication model adopted, changes introduced to the language, and how the environment was implemented, are taken into account. Other relevant issues, such as the interoperability with other Java virtual machines, portability and garbage collection (GC) algorithms will also be discussed when appropriate. The remainder of this paper is organized as follows. In Section 2, we describe the basic support for concurrent computing/programming provided by Java, as well as some other features that are relevant to understand the proposals described here. Readers that are familiar with Java's concurrency features can skip this section. In Section 3, we describe the parameters chosen to classify each of the selected projects. In Section 4, we describe the Java environments and mechanisms for supporting high-performance network-based computing that were included in this survey. Section 5 presents a classification of these systems, based on the parameters described in Section 3. Section 6 concludes this work.

2. THE JAVA LANGUAGE

Although Java is a relatively recent language, introduced in 1992, several ideas underlying the language are not original [46]: Its object model has borrowed the interface concept from Objective-C, single inheritance from Smalltalk, and some other features from Self and C++. Multithreading support can be found in some C and C++ libraries, and the Java synchronization model was created in the early 70s. The portability, obtained from the code interpretation, is not new; Basic, Smalltalk and other languages had already used this approach.

Why Java did become so popular, if it did not bring anything substantially original? Two reasons seem to have contributed to its success. First, Java's syntax is similar to that of an already known and widespread language, C++, incorporating multithreading, synchronization, and network communication, without relying on external libraries. Moreover, the design of the Java language is much cleaner than that of C++. The second, and perhaps the main reason, is the provision of features designed to help the development of Internet applications - the language's integration into browsers, and its portability are very convenient for applications that should run on an inherently heterogeneous network.

Since the proposals described in this survey make many references to Java's memory model, as well as to its support for concurrent programming and communication, we describe these features in the following sections.

2.1. Multithreading and synchronization

Programming with threads in Java is more immediate than with languages as C and C++, since Java provides a native concurrent programming model that includes support for multithreading. The package *java.lang* provides a *Thread* class that supports methods to initiate, assign priority, and verify the state of a thread. To declare a thread, for instance, the programmer just inherits the *Thread* class using the clause *extends*, as shown in line 1 of Code 1, and supplies a run method to be invoked when the thread's execution starts. The examples in this section are related to a matrix multiplication algorithm.

```
01 class mmultThread extends Thread implements GlobalVariables {
02   private parameter_t p;
03
04   mmultThread (parameter_t arg) {
05     p = arg;
06   }
07
08   void mult(int size, int row, int column, matrix_t MA, matrix_t MB,
09     matrix_t MC) {
10     int position;
11     MC.matrix[row][column] = 0;
12     for(position = 0; position < size; position++)
13       MC.matrix[row][column] = MC.matrix[row][column] +
14       (MA.matrix[row][position]* MB.matrix[position][column]);
15   }
16
17   public void run() {
18     mult(p.size, p.Arow, p.Bcol, p.MA, p.MB, p.MC);
19     /* we use a barrier here just to illustrate the use of synchronization
20         primitives, but it is not necessary. A call to join() is more
21         efficient - see subsection 2.2 */
22     try { bar.barrier(); }
23     catch (InterruptedException e) {}
24   }
25 }
```

Code 1 - A fragment of a matrix multiplication code. Each thread multiplies a row by a column. To declare a thread, the class must inherit the *Thread* class and supply a *run* method (line 17) that will be invoked when the threads start executing.

The creation of a thread follows the same process as creating an object in Java, using the *new* operator. To start the execution of a thread, the *start* method of the *Thread* class must be invoked. The example in Code 2 shows how to create and to start a thread. Methods for assigning priorities to threads are also available.

```
01 public class Mmult {
02
03   public static void main(String args[]) {
04     /* declare variables, initialize or read matrix values */ ...
05     /* Process matrix, by row and column. Create a thread to process
06         each element in the resulting matrix */
07     num_threads = 0;
08     for(row = 0; row < size; row++) {
09       for (column = 0; column < size; column++) {
10         /* set parameter p */
11         threads[num_threads] = new mmultThread(p);
12         threads[num_threads].start();
13         num_threads++;
14       }
15     }
16     /* Print results */
17   }
18 }
```

Code 2 - Another example of matrix multiplication in which a thread is created (line 11), which multiplies each row by a column of the matrix. The thread is then started (line 12).

Besides multithreading, the language also includes a set of synchronization primitives. These primitives are based on an adaptation of the classic monitor paradigm proposed in [26]. The standard semantics of Java allow the methods of a class to execute concurrently. However, the reserved word *synchronized* can be associated to given methods in order to specify that they can only execute in a mutual-exclusive fashion. The example (see Code 3), taken from [31], shows a barrier class which uses a barrier method that is synchronized, indicating that it cannot be executed concurrently. It would also be possible to declare a synchronized block inside the barrier method. Note that the barrier mechanism is used here just to illustrate the use of the synchronization primitive; in fact Java supports a join primitive that would provide a more efficient implementation.

```
01 class Barrier {
02
03   protected final int parties;
04   protected int count; // parties currently being waited for
05   protected int resets = 0; // times barrier has been tripped
06
07   Barrier(int c) { count = parties = c; }
08
09   synchronized int barrier() throws InterruptedException {
10     int index = -- count;
11     if (index > 0) { // not yet tripped
12       int r = resets; // wait until next reset
13       do { wait(); } while (resets == r);
14     }
15     else { // trip
16       count = parties; // reset count for next time
17       ++resets;
18       notifyAll(); // cause all other parties to resume
19     }
20     return index;
21   }
22 }
```

Code 3 - The Barrier class code. The method *barrier* cannot execute concurrently: this is guaranteed with the use of the reserved word *synchronized* (line 09).

As identified by the Application and Concurrency Work Group of the Java Grande Forum [28], thread synchronization introduces a potential performance bottleneck (see also next subsection), which ultimately hinders Java applications with large numbers of threads to scale.

2.2. The JVM memory model

The JVM specifies the interaction model between threads and the main memory, by defining an abstract memory system (AMS), a set of memory operations, and a set of rules for these operations. The main memory stores all program variables and is shared by the JVM threads (refer to Figure 1). Each thread operates strictly on its local memory, so that variables have to be copied first from main memory to the thread's local memory before any computation can be carried out. Similarly, local results become accessible to other threads only after they are copied back to main memory. Variables are referred to as master or working copy depending on whether they are located in main or local memory, respectively. The copying between main and local memory, and vice-versa, adds a performance overhead to thread operation.

The replication of variables in local memories introduces a potential memory coherence hazard since different threads can observe different values for the same variable. The JVM offers two synchronization primitives, called *monitorenter* and *monitorexit* to enforce memory consistency. The primitives support blocks of code declared as synchronized. In brief, the model requires that upon a *monitorexit* operation, the running thread updates the master copies with corresponding working copy values that the thread has modified. After executing a *monitorenter* operation a thread should either initialize its work copies or assign the master values to them. The only exceptions are variables declared as *volatile*, to which JVM imposes the sequential consistency model. The memory management model is transparent to the programmer and is implemented by the compiler, which automatically generates the code that transfers data values between main memory and thread local memory.

2.3. Communication

Java, as other popular languages, offers a set of tools and APIs (Application Programming Interface) for communication. Sockets, Remote Method Invocation (RMI), and an Object Request Broker (ORB) are available. In this section, we will describe RMI in detail, looking at the main aspects related to its use in high-performance applications.

2.3.1. RMI and serialization

Java's distributed object model defines a remote object as an object that allows its methods to be invoked from other JVMs running on different machines interconnected by a communication network. A remote object is fully described using Java's object interface which defines the methods that the remote object supports. The *Remote Method Invocation* (RMI) is the mechanism that allows a method to be invoked in a remote object interface (see Figure 2). This technique allows local and remote methods to be invoked using the same syntax.

In order to use RMI, the programmer must structure his/her application to obey the client/server paradigm, where a remote object represents the server and the client corresponds to the object that invokes the method. In addition, a simple programming recipe must be followed that includes inheriting a special *Remote* class and using some standard methods in the application code. A standard Java tool, *rmic*, is used to automatically generate a stub (auxiliary code), which acts as a local representative or proxy of the remote object to the client. The Java 2 SDK implementation of RMI uses reflection to implement the connection between RMI and the remote service object. In classic Remote Procedure Call (RPC) implementations, the skeleton figure performs this role. For a method invocation, the stub establishes a connection with the remote JVM, marshals the invocation parameters, waits

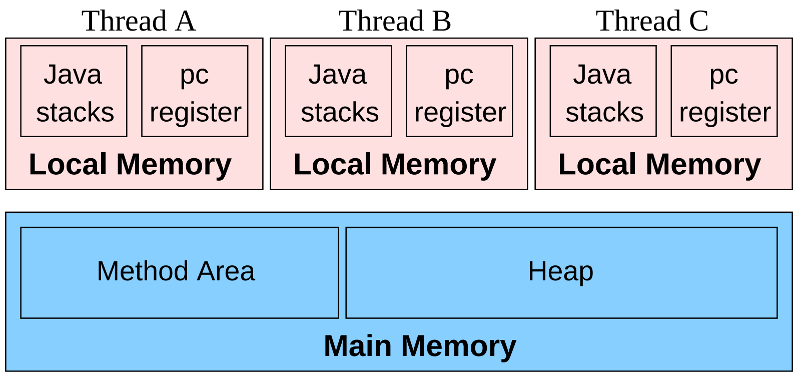

Figure 1. The internal architecture of the Java Virtual Machine's Memory

for the method invocation to complete, unmarshals all results or exceptions, and returns the outcome to the invoker.

In RMI, parameters are always passed by reference if they refer to remote objects; otherwise they are passed by copy. In the case of objects passed by copy, it is necessary to execute a serialization operation that transforms objects into arrays of bytes. Special marshaling routines will be invoked to perform the serialization operation. These routines can be written by programmers or automatically generated by the compiler. The latter option, usually preferred by the programmer, uses a structural reflection[§]mechanism that provides the appropriate byte array representation and dynamically finds the type of each object. RMI also supports polymorphism[¶], so the byte array representation must also incorporate information about the serialized types. Note that both the support for polymorphism and the use of structural reflection introduce a large performance overhead due to the large number of operations that have to be executed dynamically, which can limit its use in high-performance applications.

The Java memory model also includes an automatic garbage collection capability. The programmer does not need to worry about de-allocating objects that stop being referenced in a system. Similarly, in case of remote objects, the RMI mechanism also implements garbage collection in order to de-allocate remote objects that are no longer referenced.

2.3.2. Other aspects

RMI imposes the use of standard socket-based communication protocols, thus preventing the choice of new high-performance network protocols, such as VIA [16] and Fast Messages

[§]Structural Reflection can be defined as the "ability of a language to provide a complete reification of the program currently being executed as well as a complete reification of its abstract data types".
[¶]The use of any superclass of the subclass to refer to the instance of the subtype.

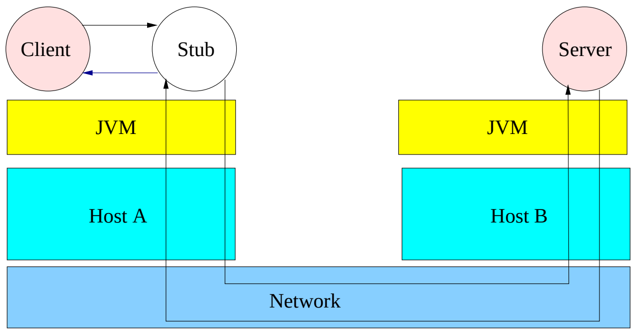

Figure 2. The RMI protocol

[40]. The inclusion of an open communication facility to the JVM, e.g., using computational reflection techniques, would add flexibility to RMI communication. In this way, a programmer would be able to employ the communication protocol most suitable for a given application [32]. The addition of a collective communication library to the language, which could provide primitives such as scatter and gather, all gather, and all-to-all, would also be useful. Although Java implements multicast collective communication via sockets, high-performance communication support would benefit from a more efficient implementation.

3. CLASSIFICATION PARAMETERS

Our classification scheme distinguishes three basic parameters relevant to the design and implementation of Java environments for high-performance network computing: (1) the model adopted for inter-process communication, (2) the modifications introduced to the language's semantics and syntax, and (3) the implementation strategy. Although there is a strong interdependence among them, explicitly or implicitly, they appear as distinguishing features in the surveyed proposals. The consideration of such aspects has allowed us to isolate important related issues and to consistently classify the projects we surveyed, producing a useful description of the alternative environments that have been proposed so far.

The way in which processes communicate constitutes an important issue in the implementation of an effective environment for high-performance network computing. Three basic approaches can be used for inter-process communication, namely distributed shared memory, message passing, or a combination of both. Parallel programs have evolved using message passing libraries, such as the Parallel Virtual Machine (PVM) [23] and the Message Passing Interface (MPI) [37], as their main method of communication. In this case, the programmer is responsible for data communication among the nodes running an application. In distributed shared memory (DSM) systems, processes share data transparently across node boundaries; data faulting, location, and movement is handled by the underlying system.

Treadmarks [30] and HLRC [51] are examples of state-of-the-art software DSM systems. Other aspects such as communication-transparency to the programmer, conformity with the language syntax, as well as the overall performance achieved, are determined by the mechanism adopted for inter-process communication.

The second aspect refers to how modifying the language impacts the overall environment, from the programmer's point of view. If the system does not introduce any modifications to both the original semantics and syntax of the language, or if only a few small changes are made, the programmer's adaptation to the new system is easier, and code reuse is also improved. We assume that a change becomes visible to the programmer if the modified environment supports a feature in a different way from Java's original specification. Features such as automatic memory management, the definition of new reserved words, and whether access to remote objects is transparent or not are related to this issue.

The third parameter is related to the strategy adopted to implement the environment, which affects code portability and performance. This survey has identified five main approaches: (1) the use of a pre-compiler, (2) modification of the Java compiler, (3) modification of the JVM, (4) extensions based on libraries written in Java, and (5) the use of native functions of a particular environment.

Further aspects, including garbage collection, and interoperability with other Java virtual machines, are also relevant. Garbage collection is particularly important, since the language specification assumes the existence of an automatic storage management system; the garbage collection mechanism has to work transparently for local and remote objects. For example, the RMI mechanism has to garbage collect remote objects that are no loger referenced. If garbage collection is ignored, the system can potentially run out of memory, since there is no statement in the language for de-allocating memory explicitly. Interoperability is a desirable feature not directly dependent on the basic language design. However, these two issues are directly related to engineering options taken in the implementation of the classification parameters considered in this survey. Although they are not included as the main classification parameters of this paper, we show how the surveyed proposals tackle these issues.

4. ENVIRONMENTS

In this section we describe several proposals which attempt to transform Java into an efficient environment for high-performance computing. Some of them tackle specific performance bottlenecks, such as the high costs associated with the standard Java communication mechanism. Other proposals are more comprehensive, trying to offer an integrated solution for application support. Whenever possible, the described proposals are grouped according to the parameters introduced in the last section. If a proposal uses more than one technique in their implementation, we consider the most significant one for classification purposes. Nevertheless, in Section 5 we make a crossover comparison taking into account all of the techniques used in each of the proposals discussed.

This section has two parts. Section 4.1 presents projects that use the Distributed Shared Memory (DSM) model for inter-process communication, whereas Section 4.2 describes projects that use the message passing model. In each section, we divide the projects according to their implementation strategy and, within each strategy, we categorize projects based on the modifications they introduce to the language's semantics and syntax. At the end of each section, we summarize the reported projects.

4.1. Inter-process communication using the Distributed Shared-Memory model

4.1.1. Java's semantics/syntax unmodified

Changes to the JVM

This section presents systems that provide a shared-memory abstraction, through transparent changes made to the internals of the basic JVM, i.e., without modifying either the semantics or the syntax of Java. Two systems fall into this class: MultiJav [14], and cJVM [1, 2, 3]. Basically, these systems differ in the way that DSM is implemented. cJVM prefers the proxy design pattern (PDP) [22] to implement the Single System Image (SSI) abstraction, whereas MultiJav implements the DSM model itself. While cJVM is an ongoing project, MultiJav has been discontinued.

MultiJav

One of the main objectives of MultiJav [14] is to maintain the portability of the Java language, allowing its use on heterogeneous hardware platforms. The MultiJav's approach is to implement the DSM model within Java, by modifying the JVM, and using Java constructs to support concurrency, thus avoiding any change to the language definition. An apparent shortcoming of MultiJav is that all of the objects are potentially shared, since the programmer cannot declare explicitly which objects are to be shared. However, the MultiJav runtime system can detect automatically which objects should be shared, catering for their consistent usage. Shared object identification is achieved through an analysis of the load/store instructions of the bytecode being executed. This technique seems to be the main contribution of the project. Different threads are permitted to access variables of the same object, thus a significant amount of false sharing may occur. MultiJav uses a multiple-read/multiple-write protocol to alleviate the potential false sharing issue.

A MultiJav program begins execution in one virtual machine, named root, but spawned threads can migrate to another machine, afterwards. In order to conform to the standard Java semantics, monitors are global to all the sites that participate in a computation. Thus, at each participating site, a monitor queue can contain local threads and remote threads; only the higher priority remote thread has to be queued. The changes in objects are detected at execution time through the use of a diff-like mechanism [30], and updates are recorded and disseminated at synchronization points.

Some implementation issues in MultiJav are still open. For instance, the use of MultiJav in heterogeneous systems requires a full implementation of the support mechanisms on each target platform in order to allow heterogeneity, but the authors do not estimate the effort required for such a task. Also, the overheads that the adopted synchronization management

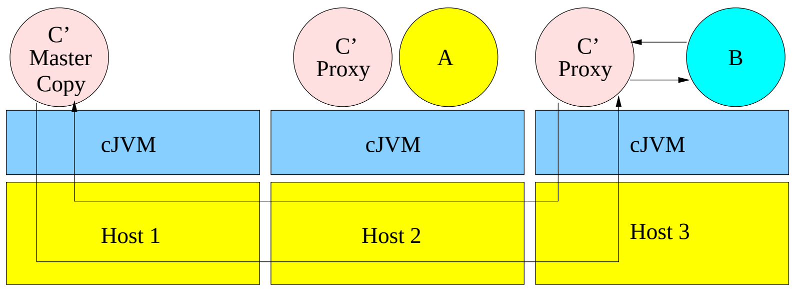

Figure 3. The master copy (MC) of object C was created at host 1. Objects A and B access object C through C's proxy. The figure shows one B's access.

mechanism introduces are unclear. Unfortunately, a performance analysis is not available for MultiJav since it has not been implemented yet.

cJVM

cJVM [1, 2, 3] has been developed at IBM Haifa Research Laboratory. cJVM supports the idea of SSI view of a cluster. In other words, the Java application will have the illusion that the cluster is a single computing resource. In cJVM (see Figure 3) a new object is always created in the node where the request was executed first. Every object has one master copy that is located in the node where the object is created; objects from other nodes that access this object use a proxy.

Aiming at performance optimization, during class loading, class methods are classified according to the way they access the object fields. Thereafter, the classification helps to choose the most efficient proxy implementation for each method. Three proxy types are supported: (1) a standard proxy which transfers all of the operations to the master copy; (2) a read-only proxy which applies the operations locally (since it is guaranteed that only the fields that never change will be accessed, the proxy can replicate and maintain these fields); and (3) a proxy that locally invokes methods without state, since these methods do not access object fields.

The introduction of proxies causes the stack of a Java thread to be distributed across multiple system threads on different machines. Thus, to ensure that programs execute correctly, cJVM treats Java calls that access the heap in a special manner. The bytecode that accesses the heap is modified so that cJVM can determine whether accesses to the data are local or remote to the node where the bytecode is executed. If data is remote, the necessary remote accesses are carried out. In order to support both remote accesses and redirection of methods, each cJVM process contains a group of threads that are responsible for receiving and serving such requests.

cJVM also modified the implementation of the new *opcode*, allowing the creation of threads in remote nodes. If the parameter for this *opcode* is a class that implements *Runnable*, then the new bytecode is rewritten, as the pseudo bytecode *remote_new*. This pseudo bytecode, when executed, determines the node best suited to create a new *Runnable* object. A pluggable load balancing function makes the choice of the best node. Notice that the thread creation approach differs from the one used for object creation, which creates the object in the node where the request was executed first.

The Portable Business Object Benchmark (pBOB) was used to evaluate performance of cJVM against that of Sun JDK1.2. pBOB was influenced by the TPC-C benchmarks [45] and consists of N warehouse composite objects that represent customers, stock items, orders, etc., which concurrently execute transactions against their warehouses. The results showed speed-up of 3.2 for four nodes, but considering that the application is highly parallel a near linear speed-up should be expected. The hardware platform used in the experiments was not described. Further performance studies are needed, especially for other classes of application, such as those described in [27].

4.1.2. Modification of Java's semantics/syntax

Changes to the JVM

Java/DSM

Java/DSM [50] under development at Rice University was the first proposal to support a shared-memory abstraction on top of a heterogeneous network of workstations. The main idea behind Java/DSM is to execute an instance of JVM in each machine that participates in the computation by using a system that combines Java portability with TreadMarks [30], a popular software DSM library. Except for the changes to Java's semantics, Java/DSM is similar to the systems presented in the last subsection (e.g., MultiJav). In contrast to those systems, however, the heap is allocated to the shared memory area, as shown in Figure 4, by using TreadMarks. Thus, the classes read by the JVM are allocated automatically to the shared memory. Two restrictions are imposed on the programmer: (1) a thread cannot migrate between machines, and (2) a thread's location is not transparent. The first restriction hinders dynamic load balancing activities, preventing thread migration from overloaded processors to idle ones, whereas the second restriction requires the programmer to be aware of each thread's location.

Java/DSM extends the Boehm and Weiser collector [9], which is a distinguishing contribution of the project. The garbage collector of each machine maintains two lists; one containing remote references for objects created locally (export list), and other keeps references to remote objects (import list). The lists contain an estimate of the actual cross-machine reference set which is used only for garbage collection purposes. Before a message is sent to another machine, the runtime DSM support invokes the garbage collector to verify if it contains valid references to local objects; these references are inserted in the export list. Likewise, incoming messages are inspected and references to remote objects are inserted in the import list. Garbage collection is performed using a weighted reference counting algorithm to decide when a reference can be discarded from the export list. For most of the time, each machine independently executes the

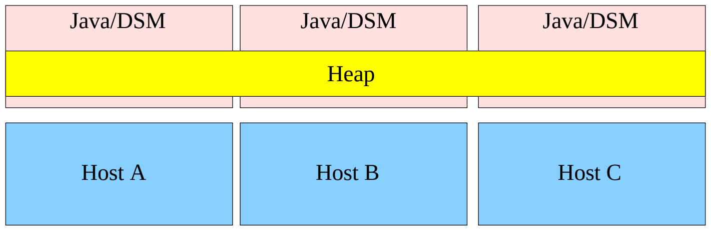

Figure 4. The heap in Java/DSM is shared among all Java/DSM nodes.

garbage collection, although some synchronization operations are required once in a while in order to take care of cyclic structures.

Since Java/DSM is intended to work on a heterogeneous hardware platform, data conversion is required. For data conversion, the data type is first identified, which in turn determines the form in which the conversion should be done. In order to perform object identification efficiently, Java/DSM requires that only objects of the same size be allocated in a given page. In addition, an extra field, which contains a pointer to the handle, is added to the object's body. Note that object representation in the Java standard includes only two components: the handle and the body. The handle contains a pointer to a structure that stores type information about all of the fields, and also a pointer to the body. Java/DSM adds a back pointer from the body to the handle. These modifications simplify the task of locating the descriptor of each object's type. More specifically, given any address, Java/DSM can promptly identify the page number and the size of the objects that the page contains. Once the beginning of the object to which the address belongs to is found, the back pointer can be followed to determine the object's type, which in turn allows the conversion to proceed quickly.

So far, Java/DSM's attempt to provide a heterogeneous software DSM has not been fully achieved. Although a preliminary comparison between Java/DSM and standard Java RMI was reported in [50] using an experimental distributed spreadsheet with support for collaborative work, the results were superficially described without presenting any performance figures.

Changes to the compiler

Jackal

Jackal [48], from Vrije University in The Netherlands, implements a software DSM abstraction on a cluster through special run-time support and an associated compiler. The compiler, which generates native code, is also used to make optimizations, such as data prefetching. Jackal requires some semantic changes to be made to Java, however.

The Jackal run-time system implements a cache coherence protocol for the memory units, called regions, where are defined as either objects or fixed-size array partitions. The coherence protocol is based on self-invalidation, in which every time a thread reaches a synchronization point it invalidates its own data, ensuring the coherence of subsequent accesses. Although such a protocol is simple, it can invalidate data that will not be touched by any other node and thus add unnecessary overhead to the coherence mechanism. To implement the self-invalidation protocol, each thread maintains a control list of the regions accessed for reading and writing since the last synchronization point. At synchronization points, cached copies in the list are invalidated and modified regions are sent to their original locations. Jackal uses a home-based coherence protocol in which the home nodes allocate regions and requests for regions are sent to their corresponding homes. To avoid unnecessary address translation, each region refers to the same virtual address across the machines. The compiler generates an access validation every time a field of either an object or an element of an array is accessed. This verification determines whether or not the region referenced by a pointer contains a valid local copy. In the case of detecting an invalid access, the runtime system contacts the home node and requests a copy of the region; the received copy is then stored at the same address location as the original in its home node. Jackal provides a memory model that differs from the Java standard. In Jackal, it is taken for granted that programs: (a) are race-condition free, (b) have sufficient synchronization declarations for concurrent read/write accesses to objects or arrays, and (c) apply such synchronization declarations to the whole object or array.

Jackal implements both local and global garbage collection based on the mark-and-sweep protocol. When a node is out-of-memory, it executes a local garbage collection. As long as the collection is made locally, no synchronization is necessary. However, the local garbage collector cannot discard objects that are referenced by other objects in remote nodes. When the number of nonlocally referenced objects becomes sufficiently large, the local GC algorithm may not be able to release enough memory, and the global GC phase is started. The main cost of global GC is due to the amount of communication and synchronization among the involved nodes.

A micro benchmark was executed to measure the overhead of garbage collection, including access validation plus latency and the throughput of object transfers. All tests were run on a cluster of 200MHz Pentium-Pros, running Linux, and connected by a Myrinet network using LFC [8] as the communication layer. The results showed that the local GC performance was worse than that of JDK whereas the relative overhead for the global GC was less than 5%. The access verification code added an overhead of 14% approximately. For transferring small objects, the average latency was 35.2 μs and throughput varied from 3.9 Mbytes/s to 24 Mbytes/s, depending on whether the compiler activates prefetching or not. Other benchmarks were also executed: SOR, Ray-tracing, and a Web server. The results showed that prefetching as generated by the compiler contributes significantly to the reduction of SOR execution time. For the Ray-tracing benchmark, both the runtime system and garbage collection generated a large overhead. The results from the Web server were not made available.

Using a Java library

This section presents systems that provide the shared-memory abstraction through the implementation of a Java library that modifies either the semantics or/and the syntax of the language. Two systems fall into this category: Charlotte [6, 29] and Aleph [25], from Brown University. Charlotte was designed to use idle computers connected to the Internet to solve large computational parallel problems. Charlotte allows any machine with a Javacapable browser to participate in any ongoing computation on the Web; the participating machines can join or leave a computation at any moment. The Aleph Toolkit is a collection of Java packages designed to support distributed computations that run across networks of heterogeneous workstations. Aleph provides the ability to start threads on remote processors. Communication is implemented using message-passing, including an ordered multicast facility, and by shared objects.

Charlotte

Programs in Charlotte [6] consist of alternate sequential and parallel steps. The application has a manager that executes the sequential steps and controls execution of the parallel steps (defined with *parBegin()* and *parEnd()* constructs). In the parallel steps, routines are defined and distributed to the workers, which are applets executing in browsers. At the end of each parallel step, a barrier synchronizes all the running routines. The memory in Charlotte is logically partitioned into private and shared segments. The shared memory has concurrent-read, exclusive-write semantics and is implemented at the data type level, with Charlotte classes corresponding to the primitive types. Sharing is object-based, with access to the shared data being made through special methods: *get()* and *set()*. In a read access, if an item is identified as invalid then a new copy is requested from the object manager. In a write access, the object is marked as updated so that all modified objects can be sent back to the manager at the end of the routine execution.

Read accesses can take a long time to request data to the manager, especially in environments with high communication latency. In order to reduce the read overhead, every read operation prefetches a group of objects from the manager instead of only one. It is up to the programmer, with the use of appropriate annotations [29], to give Charlotte in advance some information of which data will be probably used by a routine; the annotation technique is also used for write operations. The annotations resemble read/write operations used in the message-passing approach; the difference here is that the data does not need to be explicitly addressed using primitives like send/receive. Charlotte allows for the verification of annotations at runtime. However, the overhead of this verification can be transferred to the compiler. The use of annotations, the main contribution of this proposal, may be effective for improving the performance of an application. However, to take advantage of annotations, a good understanding of the application is required by the programmer. Moreover, it is unclear if irregular applications would benefit significantly from such an approach.

Additional optimizations were proposed[29] to improve Charlotte's performance: since the manager knows which data is currently valid for each worker, it can allocate to a given work routines that operate on the data already owned by that worker, thus minimizing the amount of extra data that has to be moved around. Another possible optimization is to keep intact all local data stored in a worker at the end of a parallel step (instead of invalidating the data, as Charlotte does), and overwrite this data with new values only when necessary. The programmer can declare shared variables as not modifiable in order to help the implementation of this cache-like optimization. Charlotte also provides fault-tolerance and a mechanism for adaptive parallelism.

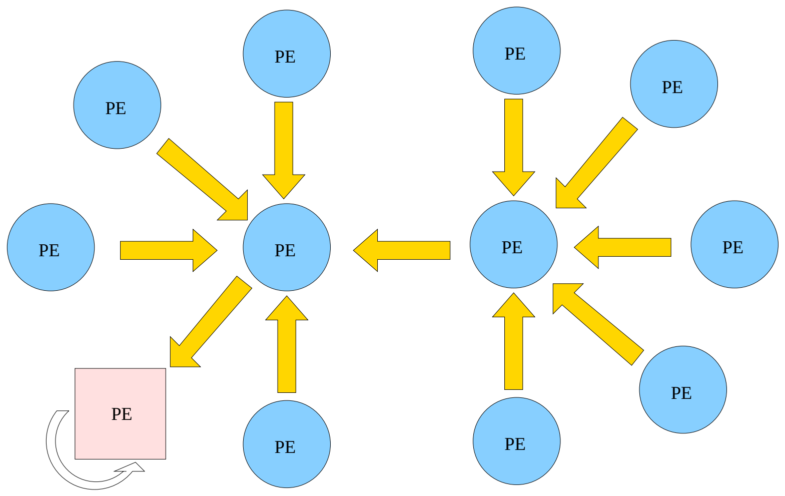

Figure 5. The Arrow Protocol. Each processor element (PE) uses a pointer to indicate the path that must be followed to reach a given object. If a PE points to itself, either the object is located within the PE or it will be shortly moved to it (represented by the square in the picture). If the link points to another PE, then the object belongs to a tree's component (adapted from [25]).

A matrix multiplication application was used as a benchmark [29] to compare several versions of Charlotte with a version of the same application based on message passing. For the version that does not verify annotations during execution time, assuming that the compiler can do this, the results indicate that Charlotte yields execution times that are competitive with those of message passing implementation.

Aleph

A distributed Aleph program executes on a number of logical processors, called Processing Elements (PE). Each PE is a JVM with its own address space. Aleph allows threads to start in remote processors, and to communicate with shared objects (with transparent synchronization and caching) or using message passing, including also an option for reliable orderly multicast. To share objects, Aleph provides the class *GlobalObject*, which allows PEs to share any serializable object. In order to use a global object, the programmer should explicitly invoke *open()* which also sets the object's access mode. A *release()* method is available to explicitly release global objects. The methods of *GlobalObject* class invoke the directory manager, which is a system object in charge of maintaining the replicated copies of distributed shared objects. Aleph implements three different directory protocols: home-based, the arrow protocol, and a hybrid protocol, which is a combination of both. In the case of the home-based protocol, an object can have both a single read/write copy and multiple read-only copies. Aleph introduces the arrow protocol (see Figure 5), which works on a spanning tree covering all the PEs. Each

PE keeps a pointer, called arrow, which points either to itself or to one of its neighbors on the PE's tree. If a PE points to itself, then either the object is located in or it will be shortly moved to that PE. Otherwise, the link points to another PE, and the object belongs to a component of the tree. Informally, it can be stated that, if a PE is not the owner of an object, then it knows in which direction the object can be found. The hybrid protocol assumes that each object has a home that only knows the last PE that requested the object. Aleph also permits the program to use the message-passing approach. Messages in Aleph are loosely modeled on active messages [19] where each message encompasses a method and its arguments, which is invoked on message arrivals. New classes of messages are defined by extending the abstract class *aleph.Message*. The programmer must then provide a *run()* method, which will be called at the receiver upon message arrival.

Results are presented for the three directory-based protocols executing an application that evaluates the time needed for a group of machines to increment a shared counter. An equivalent comparison is also made for other applications including Cholesky, Ray-tracing, and TSP (Traveling Salesman Problem). The results are presented qualitatively using bar graphs, without showing performance figures or performance comparison against sequential algorithms, which limits the analysis.

4.1.3. Summary

MultiJav, cJVM, Java/DSM, Jackal, Charlotte, and Aleph chose the distributed-shared memory (DSM) model as their main approach for inter-process communication. However, Charlotte and Aleph have some particularities: Charlotte offers to the programmer the option of using annotations that resemble read/write operations of the message-passing model. Aleph also permits message passing, although the DSM model is the main focus of the project.

MultiJav keeps Java's semantics and syntax unchanged, despite introducing the distributed shared memory model by modifying the JVM. Such an approach offers two advantages: (1) the potential of reusing standard JVM code; and (2) application performance tends to be better than using a Java-based library. MultiJav offers also an automatic mechanism for detecting shared data. The potential disadvantages of MultiJav are (a) a lack of interoperability with others implementations, and (b) less portability when compared to Java-based library implementations. In comparison with MultiJav, Java/DSM has two disadvantages: (1) Thread's location is not transparent; and (2) threads cannot migrate, which prevents dynamic load balancing. Unfortunately, both Java/DSM and MultiJav have not presented any performance figures. cJVM is similar to MultiJav since both maintain the Java semantics/syntax and extend the JVM to support DSM. However, they differ in the way that DSM is implemented. cJVM uses the proxy design pattern to implement the Single System Image abstraction. In this model, there is just one instance of an object (called the master) for all of the hosts that participate in the computation. All the other hosts access the master copy through proxies. Note that a shortcoming of cJVM is that object master copies may become potential JVM bottlenecks. On the other hand, MultiJav allows multiple copies of object instances to coexist, and uses a diffing mechanism to detect changes made to the objects at execution time and disseminating updates at synchronization points. Java/DSM uses the TreadMarks software DSM library that also implements a similar diffing mechanism.

An interesting feature of cJVM is that it enables the programmer to control system load by offering an attachable load balancing function.

Jackal combines an extended Java compiler and run-time support to implement the DSM abstraction. The Jackal compiler inserts code for access validation every time an object or array is accessed. Performance results were reported and showed that access validation is an expensive operation, with overheads around 14%. Another potential source of overhead in Jackal is the use of a self-invalidation protocol, in which data are invalidated even if they have not been modified by any other node. Also, Jackal's adopted memory model can be a source of overhead, since it forbids concurrent read or write access to different fields of a single object.

By implementing DSM using Java libraries, as in the case of Charlotte and Aleph, has some pros and cons. For instance, both favor program portability over performance gains when compared with MultiJav, cJVM, or Java/DSM. In addition, Charlotte offers the programmer the option of using annotations in the code to improve performance, which may be effective depending on the programmer's knowledge of the application, whereas Aleph introduces the arrow directory-based protocol. Unfortunately, both works reported few performance results, thus it is important that more experiments be carried out before any conclusion can be made regarding these proposed environments. Indeed, Charlotte designers might investigate whether regular applications can benefit or not from annotations, while Aleph's authors might investigate how well the arrow protocol and its hybrid version perform across several classes of applications.

4.2. Inter-process communication using the Message Passing model

4.2.1. Java's semantics/syntax unmodified

Using a native library

mpiJava

The mpiJava [5], from NPAC at Syracuse University, is a Java interface for existing MPI [37] implementations. mpiJava is made relatively simple by using Java wrappers from the Java Native Interface[‖] (JNI) to make MPI calls. However, Java requires modifications to both the syntax and semantics of several MPI functions. For instance, send and receive functions can only transfer single-dimension arrays of primitive data types. Similarly, the argument list of some functions requires some changes to accommodate the fact that in Java arguments cannot be passed by reference.

Some MPI functions omit the argument that identifies the array size, since this can be obtained through Java's length property. The MPI destructor function is called by Java's finalize method, except for *Comm* and *Request*, which have explicit *Free* members. The

[‖]JNI is a programming interface for writing Java native methods (methods used by a Java program but written in a different language) and embedding the Java Virtual Machine into native applications.

introduction of explicit calls to a method that releases memory breaks up Java's memory management semantics, since the programmer must explicitly free the allocated memory.

Some experimental results, using both models of shared memory and distributed memory, show that mpiJava adds a fairly low overhead when compared with native implementations. This result is partly due to the fact that the performance comparisons measured execution time of native code against that of interpreted code. The results were obtained using two platforms: (1) WMPI, a Windows-based implementation of MPI, running on two dual processor (P6 200MHz) Windows NT 4 workstations with 128 Mbytes of DRAM; and (2) a Solaris version of MPICH, running on two dual processor (200MHz Ultrasparc) Solaris workstations with 256 Mbytes of DRAM. The mpiJava overhead under WMPI was about 100 ms whereas the MPICH overhead was between 250 and 300 μs, which is significantly lower.

Java-to-C Interface Generator (JCI) plus MPI

The Java-to-C Interface Generator (JCI) [24] has been developed at IBM T. J. Watson Research Center. JCI is a Java to C interface generator similar to Java Native Interface. Although JCI is not intended to be an environment for high-performance computing, Java programmers can use JCI to benefit from native libraries such as MPI, to improve performance in high-performance applications. The input to JCI is a header file that contains prototypes of C functions provided by the native library. JCI then generates files with stubs for C functions, declarations of Java native methods, and scripts for compilation. JCI allows Java programmers to use native library packages, such as MPI and the ScaLAPACK linear algebra package.

Some of the mpiJava restrictions are not found in JCI, or they can be eliminated using methods and functions that are available in the JCI tool kit. For example, JCI can create a mapping between absolute and relative C's addresses; *JCI.ptr* is a method similar to the C operator *&* which is generated by JCI. Derived types, like *MPI_TYPE_STRUCT*, can also be used provided that they follow the data layout as described in the language specification. In case of multidimensional arrays, the programmer needs to adapt such structures to one-dimensional arrays.

Java's array of arrays is defined as an array of pointers to array objects instead of a contiguous two-dimensional array. Actually, an array in Java is described using *MPI_TYPE_INDEXED* rather than *MPI_TYPE_contiguous*, as it would be in C. However, the programmer has to reallocate arrays in memory in order to make them contiguous, before they can be passed to native functions. The reallocation overhead can be high for large arrays, so JCI designers represent matrices as one-dimensional arrays. In Java, it is not possible to pass array blocks only entire arrays as parameters in function calls. In order to overcome such a limitation, JCI implemented the method *JCI.section (array, index)*.

A performance comparison has been carried out [24] between C, Fortran-77 and Java linked with native libraries using two benchmarks: IS, from the NAS suite, and MATMUL, from the PARKBENCH suite. The IS benchmark runs on two different platforms: (1) a Fujitsu AP3000 (Ultrasparc 167 MHz nodes); and (2) an IBM SP2 system (120 MHz P2SC processors) using a IBM's port of JDK 1.0.2D, the IBM Java compiler hpcj, which generates native RS/6000 code, and Toba 1.0.b6, which translates Java bytecode into C. The MATMUL benchmark runs on a Sparc workstation cluster and on an IBM SP2 system (66 MHz Power2 "thin1" nodes). The results showed that Fortran-based MATMUL outperforms Java by 5% to 10% whereas Java-based IS programs were twice as slow as the C versions. One explanation is that in MATMUL most of the performance-sensitive calculations were performed by the native code.

Other

Several other projects describe Java binds either to MPI or to PVM. Two distinct Java binds to PVM [20, 44] share the same name: JPVM. JPVM [20] developed at University of Virginia presents a simple front-end to PVM. JPVM [44] developed at Georgia Institute of Technology is an interface that was developed using features of native methods and allows Java applications to use PVM. MPIJ [18] is a Java-based implementation of MPI integrated with DOGMA (Distributed Object Group Metacomputing Architecture). JMPI [17] is an MPI environment built on top of JPVM [44]. JavaWMPI [35] is a MPI version built on MPI for Windows. Although these projects present some contributions, we do not classify then mainly because they are conceptually similar to mpiJava.

4.2.2. Changes to Java's semantics/syntax

Modification of the Java compiler

Manta

Manta [47, 33, 39, 34] is a Java system for high-performance computing that uses a native compiler to translate from Java directly to executable code. A disadvantage of Manta is that some changes were introduced to the semantics and syntax of Java. Besides compilation, Manta tackles the three main sources overheads in Java: serialization, RMI streams and dispatch, and the network protocol. For serialization, Java uses a structural reflection mechanism to determine at run time the type of each parameter passed within remote calls. The idea behind Manta is that most of the serialization/un-serialization codes can be generated at compile time, thus reducing the overheads of dynamic inspection. Manta's protocol for serialization also provides some optimizations. For example, in case of an array of primitive types, a direct copy from memory to a message buffer is made, avoiding the traversal of the whole array. A hash table is also used to keep serialized objects. Manta attempts to improve program performance by decreasing the number of layers used originally in the RMI protocol, so that fewer operations are required for parameter copy and method calls. As an example, the parameters of a remote invocation are directly copied to a buffer, while in the RMI protocol several copies need to be made. Another important factor that contributes to Manta's performance is the fact that Manta's runtime system is written in C, while all the Java RMI layers are mostly interpreted. Figure 6 compares the layer's organization of the protocols. Finally, RMI uses the TCP/IP protocol while Manta relies on a more efficient protocol. Manta's choice was Panda[8], a user level communication library that has independent interfaces to both the hardware and network protocol.

Manta uses a modified RMI protocol that might cause interoperability problems with other JVMs. Manta's solution is to let the compiler generate both bytecodes and native code. The

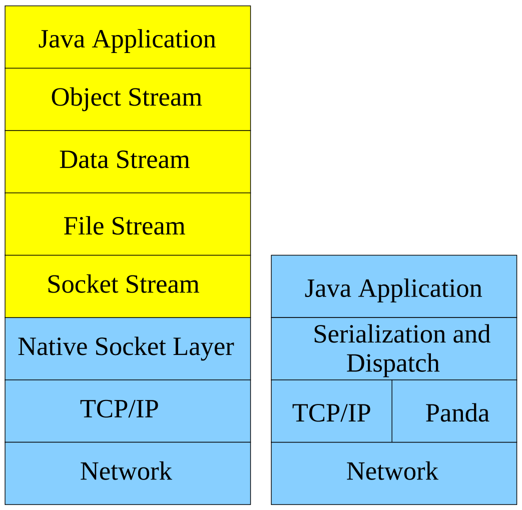

Figure 6. The structures of Sun (leftmost) and Manta RMI protocol. Layers in dark denote compiled code (adapted from [33]).

former is placed in a HTTP server, allowing remote JVMs to access them. When treating standard RMI calls, a Manta machine access the required bytecode, and dynamically compiles and links it to the application.

In Manta, the RMI protocol and the garbage collector work together by maintaining the reference paths made by the computation nodes. Manta uses a local garbage collector based on the mark-and-sweep algorithm [36]. Each computation node executes its local garbage collector, using a dedicated thread that is activated by either the runtime system or the user. A distributed garbage collection is implemented across the local garbage collectors, using the mechanism of reference counting [15] for remote objects.

The performance impact of all the above described optimizations appears when Manta is compared with JDK. Experiments were run on a homogeneous cluster of 200 MHz Pentium Pro processors, each with 128 Mb of memory and running linux (kernel 2.0.36). The cluster nodes were connected by two networks: Myrinet (1.2 Gbit/sec) and Fast-Ethernet (100 Mbit/s). For Myrinet, using the simplest remote call with no input or output parameters, the Manta latency is 37 μs against 1316μs for Sun JIT (Blackdown) 1.2 and 550 μs for IBM JIT 1.1.8, respectively. For Fast-Ethernet, the Manta latency is 207 μs against 1500μs and 720 μs for Sun JIT (Blackdown) 1.2 and IBM JIT 1.1.8, respectively. Manta also performed better than the Sun RMI protocol for other benchmarks [34]. Manta has also been used in a metacomputing environment [39] based on homogeneous cluster.

Manta modified the syntax of Java by introducing the reserved word *remote*, which permits the programmer to indicate which classes can be remotely invoked, replacing the language standard mechanism that requires the inheritance of the class *java.rmi.server.Unicast.RemoteObject*. This new operator provides support for the creation of objects in remote machines. However, these syntax modifications prevent Manta from reusing code written for other Java machines. Other Manta's characteristics can limit its utilization, too. For instance, some of Java's characteristics were omitted to optimize the RMI protocol, by arguing that they could reduce performance and were not necessary for high-performance computing. The shortcomings of such an argument are twofold: the programmer cannot reuse old code, and he/she has to adapt to the Manta programming style. Another restriction is that all processes that participate in the computation should start at the same time. Furthermore, Manta does not support the heterogeneity and safety of the Java model.

Using a pre-compiler

HPJava [49, 12] and JavaParty [41] propose changes to Java' semantics and syntax and developed pre-compilers to support the changes they introduced. Although providing a versatile message passing facility, HPJava's main objective is to support the SPMD programming model. JavaParty takes advantage of object locality and provides a modified RMI facility.

HPJava

HPJava focuses on the potential benefits of including some characteristics of High-Performance Fortran, such as the distributed array model, array intrinsic functions, and libraries, all of which could make Java an attractive language for programming under the SPMD model. In particular, HPJava supports distributed arrays as a language primitive and distributed control constructs to facilitate access to the elements of a local array. Under this model, programmers do not need to know the physical location of any particular array element. To achieve this, HPJava introduces three new classes: *Group*, which defines a group of processes to which the elements of an array are distributed; *Range*, which describes the extent and mapping of an array dimension to the dimension of processes; in other words, *Range* maps an interval of integers to the dimension of processes according to a given distribution function; and *Location*, which is an abstract element of *Range*, hence *Range* can be considered as a group of *Locations*. In addition, two further classes are defined: *Subrange* and *Subgroup*, which define sub-ranges in the *Range* and *Group* objects, respectively. A distributed array is declared using the symbols "[[" and "]]" and by passing objects of the class *Group* and *Range*, or their subclasses, as parameters.

Three control operators, namely *at*, *on*, and *over*, enable distributed execution where each process executes correctly on a particular subset of a distributed array. The control operator *on* defines group of processes that share an active thread of control. For example, *on (p)* {...} defines all the operations on the distributed array that are executed by the *p* group. The *at* operator is similar to *on*, except that its body is only executed on the processor that owns a specific location. As an illustration, consider the following piece of code: *Location i = x[13]; at (i)* {...}, the commands within the body of *at* are executed on the processor that owns location 13 of the array *x*. The *over* operator implements a distributed parallel loop.

Collective communication libraries are supplied to ease the HPJava programmer's task of controlling the movement of data in a distributed environment. Some basic library commands are described, as follows. The *copy* command allows elements of a distributed array to be copied to another array, independently of the distribution format. The *remap* command copies a group of elements and redistributes them to another distributed array. The elements can be distributed to the same group of processes or distributed to a different group. The *shift* command moves a certain number of elements of a given dimension either in a cyclic or off-edge way. The *writeHalo* command supports *ghost* regions in the communication, allowing a reduction in the amount of memory copies required and the communication traffic during collective communication. Communication is deadlock free and new communication libraries can be integrated into the supplied library. The HPJava environment comprises a pre-compiler and runtime libraries. For performance evaluation purposes, Cholesky and Jacobi applications were used, but the performance results that were reported are unclear.

JavaParty

JavaParty [41] supports distributed parallel programming in heterogeneous clusters by extending Java with a pre-processor and a run-time system. JavaParty implements a shared address space in such way that local and remote accesses to both methods and variables are identical. The main change to the language was the introduction of a new reserved word, called *remote*. The Remote operator allows programmers to indicate which classes and threads should be distributed across machines within a cluster. It is not necessary, however, to indicate in which machine an object will reside, nor the communication mechanism to be used between objects. The run time system and compiler are responsible for these tasks, as well as for dealing with network exceptions caused by the communication system. The distribution of objects and threads is implemented by the run time system using the *strategy* design pattern[†], which can also be modified at run time. Another alteration introduced to the language is the permission for static methods and static variables to be remotely accessed, which is not allowed with standard RMI. JavaParty's pre-compiler generates two classes for each remote object that declares static items: one that maps instance variables and instance methods, and another for class variables and class methods (static). A third class is generated to give the programmers transparent access to the two generated classes; this class has the same name and interface as the class originally declared by the programmer.

The run time system implements load balancing, and monitors the interactions between objects. This allows the run-time system, or the programmer, to migrate objects to increase their locality. In contrast, the locality of objects is completely ignored by the standard Java execution environment. For instance, when using RMI, if one of two objects invokes a method at the other, Java activates the RMI mechanism even if they are in the same machine. In the same situation, JavaParty will check the object location and a local method invocation will be made. The local method invocation takes 0.7μs against 2.8ms for a RMI invocation, thus the penalty for using RMI unnecessarily can be very high, up to 4000 times slower. However, JavaParty does not report any performance results for applications benchmarks.

[†]The Strategy design pattern is a behavioral pattern that provides a way to select from multiple, related algorithms to accomplish a task.

Using a Java library

Java//

Java// [11], developed at the INRIA in France, is a Java library for sequential, distributed, and multithread programming, which requires no modifications to the Java execution environment. Java// resorts to concepts such as reification from Computational Reflection[‡] and Proxy Design Patterns [22] in order to ease its implementation. To implement active objects, the designers of Java// introduced a new mechanism based on two components: (1) a request queue for each object, where pending invocations can be stored; and (2) a thread for managing the queue. The object, which is the owner of a request queue, is called *Body*. Thus, Java// changes the semantics of the standard Java object. Pending requests are executed asynchronously, and the execution order depends on the selected synchronization policy. The body object follows a FIFO behavior if the programmer provides no policy. Java// offers three mechanisms to declare active objects: (1) calls to the *Java//.newActive* method, which extended the *new* functionality to allow the declaration of active objects; (2) calls to *Java//.turnActive* which transforms a passive object already declared into an active object. Note that both *Java//.newActive* and *Java//.turnActive* offer the programmer the option of creating the object in a remote node. The third mechanism implements the *Active* interface.

The concept of future objects is applied for inter-object synchronization. A future object is simply "a placeholder for the not-yet-performed method invocation" [11]. In this way, the thread that made the invocation can continue its execution as long as it does not need to invoke methods in the returned object, in which case the invoking thread blocks automatically. This concept is transparent to the programmer, so no change to the invoking thread is required. A future object is created whenever a method is invoked in an active object. This principle is known as wait-by-necessity, and synchronization is data-driven. In some situations, however, this synchronization type is not used, for example, when the return type is either primitive or final. There are situations in which the synchronization is not directly tied up to the invocation of a method within an object. In such cases, two other methods are available: *Javall.wait(obj)*, which explicitly waits for the object *obj*, and *javall.isAwaited(obj)*, which returns a boolean value to indicate whether the object *obj* has reached its synchronization point. The latter method allows a thread to carry out other useful tasks while waiting for the object *obj*.

For each class, Java// centralizes intra-object synchronization within a special method called *live*. If the class does not provide this method, the *Body* queue manager uses its own standard live method, which obeys the FIFO policy. If a class implements *Active* (to turn an object active), the programmer can overwrite the standard policy by managing the request queue explicitly through the methods *serveOldest()*, *serveOldest(method met)*, *serveOldestBut(method met)*, and *waitARequest()* implemented by the *Body*. For implicit synchronization the method *forbid (method, condition)* is used, which works as a logical guard impeding the access to the method *method* when the condition *condition* is true.

[‡]Behavioral (or Computational) Reflection can be defined as "the ability of the language to provide a complete reification of its own semantics (processor) as well as a complete reification of the data it uses to execute the current program".

For illustration purpose, two applications, namely matrix multiplication and a collaborative application that uses a Ray-trace algorithm, were described in the Java// paper [11] though performance results were not presented.

UKA-Serialization and KaRMI[§]

Serialization and Java RMI are two main sources of overheads that the UKA-Serialization [42] and KaRMI [42] approaches attempt to reduce, respectively. Although similar to Manta objectives, these approaches are radically new in the sense that no compiler support is required and the resulting code is portable since UKA and KaRMI are written entirely in Java. The central idea is that programmers replace both the serialization mechanism and the remote invocation of the language by library calls that implement several optimizations to improve performance. UKA-Serialization tackles the serialization problem on four different fronts: type coding, internal buffering, buffer accessibility, and maintaining type information upon hash table reset. The type-coding problem happens because Java has to keep enough information on persistent objects stored on disks so that the objects can be retrieved later, even if the bytecode used originally to instantiate objects has been discarded. In general, parallel programs executing on clusters do not require high degree of persistence, because objects' lifetimes are often shorter or equal to the task execution time and that all nodes in a cluster have access to the same bytecode, through the common file system. Given that the UKA-Serialization uses a textual form to represent classes and packages, the type coding is simplified and the serialization performance is improved significantly. Each remote method invocation should begin with a clean hash table so that objects which are re-transmitted will hold their new states. To implement this property, one of two alternatives can be used. Either create a new serialization object for each method invocation or call the *reset* method in the serialization object. The effect of both is to clean the information, including type information, of all objects that were previously transmitted. Therefore UKA-Serialization creates a new *reset* method which cleans only the hash table, maintaining the type information intact.

The implementation of serialization in JDK presents some problems related to the use of buffers. First, JDK implements buffered streams on top of TCP-IP sockets and the receiver does not implement any buffering strategy, thus it ignores the number of bytes required to perform marshaling operations to objects. The authors of UKA-Serialization argue that this approach is too general since it does not explore any knowledge about the number of bytes of the object representation. In contrast, the UKA-Serialization handles the buffer internally so as to take advantage of the object representation. Moreover, the optimized buffering strategy reads all the bytes of an object at once. Second, buffer access is inefficient in that JDK buffers are external, thus if a programmer wants to write directly into a buffer he/she must use special writing functions, which incur overheads. The UKA-Serialization itself implements the required buffering, thus avoiding additional method invocation. By making the buffer public, UKA-Serialization enables marshaling routines to write data straight to the buffer.

[§]Although KaRMI and JavaParty were developed by the same research group, they can be used as distinct tools.

KaRMI tries to improve performance of the Java RMI through several optimizations. For instance, its interface between RMI layers offers two improvements: (1) the RMI invocation requires just two additional invocations, and (2) a more efficient implementation of the transport layer. As a result, each KaRMI remote call creates just one object against 26 objects in the RMI. Similarly, KaRMI executes native code when interacting with device-drivers, whereas RMI makes two calls to native methods for each argument or return values different from *void*, and five more native calls for each remote invocation.

Experimental results show that UKA-Serialization reduces object serialization time by 76% as much as 96% in some cases, when compared with JDK serialization. For KaRMI, three classes of benchmarks were used: (1) kernels that test RMI calls between two nodes; (2) kernels that test the overload of the server from calls issued by several clients; and (3) specific applications, such as the Hamming problem, Paraffin Generation, and SOR. The benchmarks were executed on two different hardware platforms: (1) two PC Pentium II 350Mz, running Windows NT 4.0 Workstation with JDK1.2 (JIT enabled), isolated from the LAN, and connected to each other by Ethernet; and (2) a cluster of 8 DECs Alpha 500MHz, running Digital UNIX and with JDK1.1.6 (regular JIT), connected by Fast Ethernet. For small size arrays, the results show that KaRMI outperforms RMI by between 41% up to 84%. However, for large size arrays (e.g., 5000 elements), their performances are equivalent. Unfortunately, the authors do not compare their results with similar projects that have been described in the literature.

Some restrictions apply to both UKA-Serialization and KaRMI. If the computation needs persistent objects, the serialization optimization cannot be applied. Programs that use socket factory or port numbers are not supported due to the restructuring of interfaces promoted by KaRMI. Other minor restrictions are imposed too.

Using native libraries

Javia

VIA (Virtual Interface Architecture) is an emerging industry standard developed at Cornell University for user-level network interface and Javia [13] is a VIA interface for Java. VIA allows programmers to explicitly manage resources (e.g., buffers and DMA) of the network interface to directly transfer data to/from buffers located in the user's address space. Although Javia is not a complete proposal, it can be integrated into an environment for high-performance computing.

Javia consists of a group of Java classes that implement an interface to a native library. The Java classes offer interfaces to commercial VIA implementations and are accessed through the native library. Javia proposes two levels of interfaces for VIA. The first level, called Javia-I, manages the buffers used by VIA using native code, therefore hiding them from Java. Javia-I adds a copy operation on data transmission and reception, since the data should be moved from Java arrays to the buffers executing native code and vice-versa. On data transmission the copy operation can be optimized through array declaration made on the fly. Two types of calls are available: synchronous and asynchronous. An advantage of Javia-I is portability, as it can be implemented in any JVM that supports the JNI native interface.

Experimental results show that Javia-I is only 10% to 15% slower than the equivalent C code, when running on two-450MHz Pentium-II Windows 2000 beta3, using two Giganet 1.25Gbps GNN1000 interfaces cards connected through a Giganet GNX5000 switch. The second level, Javia-II, permits the programmer to manage directly the communication buffers, so that an application's specific information can be exploited to implement a better buffering policy. Buffer management is carried out using the *viBuffer* class and its methods, which only provide asynchronous primitives. This class is allocated out of the Java heap and is not affected by the garbage collector. Such buffers are accessed in a similar fashion to the Java primitive arrays, allowing Java applications to directly transmit and receive arrays and eliminating the need for additional buffers within native code. However, the programmer must explicitly de-allocate the buffers after using them. It could be argued that such buffers violate Java safety, because a programmer can waste all the memory space by simply forgetting to de-allocate the buffers. The Javia authors assume that this is not a new problem since the language does not provide any mechanism to prevent that from occurring. A shortcoming of such an argument is that a faulty Javia program, that does not de-allocate buffers, could work correctly if Java's memory management semantics were followed, allowing the garbage collection of unused buffers. In spite of this, Javia-II can be a valuable resource for communication-critical applications. Benchmarks results showed that Javia-II performance is on average 1% slower than C for message sizes larger than 8k bytes when running on the platform described above.

4.2.3. Summary

mpiJava, JCI plus MPI, Manta, HPJava, JavaParty, Java//, UKA-Serialization, KaRMI, and Javia have chosen message-passing as the model for inter-process communication. mpiJava is a Java interface for existing MPI implementations, which requires several modifications to both the syntax and semantics of several MPI functions. JCI, which has fewer restrictions than mpiJava, is a Java to C interface generator that allows programmers to benefit from existing native libraries such as MPI.

HPJava introduced the SPMD model to Java. HPJava provides classes to work with distributed arrays and distributed control constructs that enable each process to execute a particular array subset. HPJava also provides collective communication libraries to control the data movements. Some of the collective communication libraries are attractive and could be integrated to Java.

JavaParty introduces a new reserved word, *remote*, that indicates which classes and threads should be distributed across machines within a cluster. JavaParty run-time system implements load balancing and monitors the interactions between objects. In this aspect, JavaParty is similar to cJVM since both offer the programmer the option to control load balancing within the system. Another interesting characteristic of JavaParty is the ability to remotely access methods and static variables, which is not permitted in traditional RMI. Unfortunately, JavaParty does not present any performance figures. Java// also introduces new concepts to the language: (a) active objects, (b) future objects, and (c) new methods for intra-object and inter-object synchronization. The concepts behind future objects and proposed synchronization methods are powerful enough to be considered as a desirable extension to the language. Unfortunately, no performance evaluation of Java// has been presented.

Manta translates Java programs directly to executable code and tackles three main sources of overheads in Java: (1) serialization, (2) RMI streams and dispatch, and (3) network protocol. The performance comparison between Manta and the original JDK mostly favors Manta. Moreover, Manta should also outperform all the other systems presented in this survey. Despite modifying the RMI protocol, Manta can interoperate with other Java Virtual Machines. However, Manta has some disadvantages: (a) some of the Java characteristics are omitted in order to optimize Manta's implementation, (b) all of the processes that participate in computations must start at the same time, and (c) some Java features such as portability and built-in verification checks that make the language attractive for developing large and complex systems are not available in Manta. UKA-Serialization and KaRMI cope with serialization and RMI, using a different implementation approach. No compiler support is required since both KaRMI and UKA-Serialization codes are written in Java, making them portable and platform-independent. Javia is another attempt to tackle the poor communication performance of Java. Javia performance is excellent - just 1% slower than C for the benchmarks tested. However, the programmer must manage the memory allocated for communication, thus violating Java's automatic memory management.

5. CLASSIFICATION

Table 1 summarizes all of the proposals and environments described so far, using the classification parameters established in Section 3.

The semantics/syntactic changes column refers to whether the changes to the language made by the project are perceived or not by the programmers. In regard to cJVM our classification assumes that the change in the *new* opcode is transparent to the programmer. As this paper has made clear, the majority of the proposals modify the semantics and/or the syntax of Java. This happens because: (a) some proposals add new capabilities to Java, such as Aleph's declaration of global objects; (b) some proposals try to ease the programmer's task, such as Manta's and JavaParty's *remote* keyword; or (c) some project decisions forced the semantics/syntax to change, such as Javia's need of explicit buffer de-allocation by the programmer. Only four of the surveyed systems do not change Java's semantics and syntax. Two of them, cJVM and MultiJav, transparently modify the JVM; the mpiJava library uses JNI in its implementation, so that JNI is not exposed to the user; and JCI, which is a JNI-like mechanism usually familiar to skilled programmers.

Most of the systems use the message-passing model for inter-process communication, since this model is more widespread than the shared-memory one in high-performance network computing. However, in a Java context, it can be argued that the latter is more natural because the adopted thread model assumes that memory is shared among all the threads. Besides, existing multithreaded programs can potentially run without modifications on a distributed shared-memory environment. Nevertheless, a potential disadvantage of DSM is that this model often performs worse than the message-passing model. Some of the systems we have described, namely Aleph and Charlotte, adopt multiple approaches in their implementation for inter-process communication. Aleph's inter-process communication offers both shared-memory and message passing options. Although Charlotte is focused on shared-memory, the optional annotations that can be made in the code resemble the message-passing approach. Moreover, in the majority of the systems described in this paper, the programmer can use message passing within a shared-memory environment, since libraries and communication mechanisms, like sockets and RMI, are also available. However, in our classification we restrict ourselves to the main inter-process communication the author focused on.

Interoperability with other virtual machines is intimately related to the approach that is chosen for environment implementation. If the proposal is implemented through a library, the interoperability with other JVM implementations is possible. On the other hand, if the proposal is implemented through the modification of the JVM or the creation of a new compiler, interoperability is potentially unfeasible. Indeed, the Manta implementation uses a modified compiler that can generate either optimized native code or standard Java bytecode, the latter capability caters for interoperability with standard machines. Thus, excluding Manta, UKA-Serialization and KaRMI, Table 1 shows that the systems that do not provide interoperability either modify the JVM or the compiler.

As stated in Section 3, automatic storage management is also an important issue for supporting a distributed Java execution environment. Similar to the interoperability issue, garbage collection is also related to the approach chosen for the environment implementation. Again, if the proposal is implemented through a Java-based library, the garbage collection is automatically done by the JVM. On the other hand, if the proposal is implemented through the modification of the JVM, it is probable that the original garbage collection must be modified too. Some of the environments do not mention how they treat the garbage collection issue. In the table, such systems are indicated by a question mark. Usually, systems that are implemented as libraries do not modify the JVM's algorithm. The only exception is Javia, which implements its own garbage collection algorithm.

6. CONCLUDING REMARKS

In this paper we have described and classified fourteen Java-based proposals for high-performance network computing. Our classification scheme has focused on three relevant aspects, namely: the adopted model for inter-process communication, changes introduced to the semantics and syntax of Java, and how each specific implementation has been carried out. We assert that a change becomes visible to the programmer if the proposed environment introduces a new feature or modifies a feature that the Java framework already provides. Further related issues, such as the interoperability with other Java virtual machines, and garbage collection algorithms have been also addressed.

The majority of the proposed systems described in this work have chosen to modify the semantics and/or the syntax of the language using the message-passing model for inter-process communication. Nevertheless, there is no clear trend about how proposals are implemented. In principle, the development of a Java-based system for high-performance network computing should not modify the characteristics of the language that have made Java popular and widely used. The use of a compiler that translates Java code into native code is an interesting solution to improve performance, although this impairs portability. Just-in time compilation or approaches like that used by Manta can improve performance while maintaining the code portability. In addition, modifications in the language may be acceptable if the performance improvement is notable. For example, Manta tackles Java's serialization problem in an interesting way, passing all the overhead of dynamic inspection to the compiler. The introduction of new features in the Java Environment could also be advantageous, such as the addition of a message passing library following MPI or PVM styles. Finally, we suggest that future projects can take advantage of the extra information that is available in the JVM's bytecode and state, in order to optimize the execution of parallel applications.

We have also highlighted some potential Java related issues that can affect the performance of network-based applications. More specifically, (a) the performance of thread synchronization primitives; (b) the data serialization operation for communication, which determines the type of the parameters to be transmitted at run-time and therefore incurs a considerable but unnecessary communication overhead; (c) the use of standard socket-based communication protocols, thus preventing the choice of new high-performance network protocols; (d) run-time checks for null-pointers and array bounds performed by the JVM; (e) the dynamic nature of execution, which prevents optimizations; and (f) the way multidimensional arrays are implemented in Java, as n-dimensional rectangular collections of elements, which makes alias disambiguation difficult and prevents other optimizations [38].

A general concern is the lack of performance measures related to the great majority of the proposals. This may be explained by the fact that some of the systems are (a) basic proposals, or (b) in an initial implementation stage. Unfortunately, even systems which are in an advanced implementation stage, and have reported some interesting experimental results, tend to use their own benchmarks in an ad-hoc fashion. This fact makes more meaningful comparison between their findings difficult. Therefore, we expect that future works in this research area will promote the use of a common benchmark suite, particularly the one that has been developed by the Java Grande Forum Application and Concurrency Working Group [27]. This benchmark is divided in three sections: (1) low-level operations; (2) kernels of application, such as FFT and SOR; and (3) large scale-applications, such as Ray-tracer and Monte Carlo simulation. Additional benchmarks to measure the associated costs of thread synchronization and remote communication should also be developed. For example, it would be useful to measure the costs of write and read operations between thread local memory and main memory, as well as, the contention during accesses to the main memory. Furthermore, the addition of Internet related kernels and applications to the benchmark would be valuable, since Java has been increasingly used in Internet Applications.

ACKNOWLEDGEMENTS

We are thankful to Inês de Castro Dutra, Vitor Santos Costa and Vinod Rebello for their helpful comments on previous versions of this paper. The comments and questions of the anonymous referees have greatly contributed to improving the quality of this survey. This research was partially sponsored by the Brazilian funding agencies: Capes, CNPq, FAPERJ and Finep.

Table I. Classification

Environment / Proposal	**Seman./Syntactic Changes**	**Implementation**	**Inter-process Communication**	**Other Issues**	
				Interop.	**Garbage Collect**
MultiJav	No	JVM Modification	Shared Memory	No	?
cJVM	No (Internally: modification of *new* opcode semantic; a new object and memory model)	JVM Modification	Shared Memory	No	?
Java/DSM	Yes (*threads* location)	JVM Modification	Shared Memory	No	New
Jackal	Yes (new memory model)	New Compiler	Shared Memory	No	New
Charlotte	Yes (use of *parBegin()* and *parEnd()* constructs)	Java Library	Hybrid	Yes	JVM-based
Aleph	Yes (object definition and access, active messages)	Java Library	Shared Memory or Message Passing	Yes	JVM-based
mpiJava	No	Native Library	Message Passing	Yes	JVM-based
JCI	No	Native Library	Message Passing	Yes	JVM-based
Manta	Yes (new reserved word *remote*, Java security model unsupported)	New Compiler	Message Passing	Yes	?
HPJava	Yes (distributed array, new reserved words: *on*, *at* and *over*)	Pre-compiler and Java Library	Message Passing	Yes	JVM-based
JavaParty	Yes (new reserved word, *remote*)	Pre-compiler and JVM Modification	Message Passing	No	?
Java//	Yes (different object view, synchronization, asynchronous call)	Java Library	Message Passing	Yes	JVM-based
UKA-Serialization and KaRMI	Yes (persistent object unsupported, *socket factory* or port numbers)	Java Library	Message Passing	No	JVM-based
Javia	Yes (buffer deallocation)	Native and Java Library	Message Passing	Yes	New

REFERENCES

1. Aridor Y, Factor M, Teperman A. 1999. cJVM: A Single System Image of a JVM on a Cluster. In *Proceedings of the International Conference on Parallel Processing 99*; Wakamatsu, Japan, September 1999.
2. Aridor Y, Factor M, Teperman A. 1999. cJVM: A Cluster Aware JVM. In *Proceedings of the First Annual Workshop on Java for High-Performance Computing in conjunction with the 1999 ACM International Conference on Supercomputing*; Rhodes, Greece, June 1999.
3. Aridor Y, Factor M, Teperman A, Eilam T, Schuster A. 2000. A High Performance Cluster JVM Presenting a Pure Single System Image. In *Proceedings of the ACM 2000 JavaGrande Conference*; San Francisco, USA, June 2000.
4. Arnold K, Gosling J. *The Java Programming Language*, First Edition. Addison-Wesley, 1996.
5. Baker M, Carpenter D, Fox G, Ko S, Lim S. 1999. mpiJava: An Object-Oriented Java Interface to MPI. In *Proceedings of the 1st Java Workshop at the 13th IPPS & 10th SPDP Conference*, Puerto Rico, April 1999. LNCS, Springer Verlag: Heidelberg, 1999.
6. Baratloo A, Karaul M, Kedem Z, Wyckoff P. Charlotte: Metacomputing on the Web. In *Proceedings of the 9th Int. Conf. on Parallel and Distributed Computing Systems*; Dijon, France, September 1996.
7. Baratloo A, Karaul M, Karl H, Kedem Z. An Infrastructure for Network Computing with Java Applets. 1998. In *Proceedings of the ACM 1998 Workshop on Java for High Performance Network Computing*; Palo Alto, USA, February 1998.
8. Bhoedjang R, Rhl T and Bal H. 1998. Efficient Multicast on Myrinet Using Link-Level Flow Control. In *Proceedings of the Int. Conf. on Parallel Processing*; Minneapolis, August 1998, pp. 381-390.
9. Boehm H, Weiser M. 1988. Garbage Collection in an Uncooperative Environment. *Software: Practice and Experience* 1988; 18(9):807-820.
10. Cappello P, Christiansen B, Ionescu M, Neary M, Schauser K, Wu D. 1997. Javelin: Internet-Based Parallel Computing Using Java. *Concurrency: Practice and Experience* 1997; 9(11):1139-1160.
11. Caromel D, Klauser W, Vayssière J. 1998. Towards Seamless Computing and Metacomputing in Java. *Concurrency: Practice and Experience* 1998; 10(11-13):1043-1061.
12. Carpenter B, Zhang G, Fox G, Li X, Wen Y. 1998. HPJava: Data Parallel Extensions to Java. *Concurrency: Practice and Experience* 1998, 10(11-13):873-877.
13. Chang C, von Eicken T. 1999. Interfacing Java to the Virtual Inteface Architecture. In *Proceedings of the ACM 1999 Java Grande Conference*; Palo Alto, USA, June 1999.
14. Chen X, Allan V. 1998. MultiJav: A Distributed Shared Memory System Based on Multiple Java Virtual Machines. In *Proceedings of the 1998 International Conference on Parallel and Distributed Processing Technique and Applications*; Las Vegas, USA, July 1998.
15. Collins G. 1960. A Method for Overlapping and Erasure of Lists. *Communications of the ACM* 1960, 3(12):655-657.
16. Compaq Corporation, Intel Corporation, Microsoft Corporation. 1997. Virtual Interface Architecture Specification. Version 1.0. *http://www.viarch.org*. Accessed on June 07, 2001.
17. Dincer K, Ozbas K. 1998. jmpi and a Performance Instrumentation Analysis and Visualization Tool for jmpi. In *Proceedings of the First UK Workshop on Java for High Performance Network Computing*; Southampton, UK, September 1998.
18. DOGMA. *http://dogma.byu.edu/*. Accessed on June 07, 2001.
19. Eicken T, Culler D, Goldstein S, Schauser K. 1992. Active Messages: A Mechanism for Integrated Communication and Computation. In *Proceedings of the 19th Annual Int'l Symp. On Computer Architecture*; May 1992.
20. Ferrari J. 1998. JPVM: Network Parallel Computing in Java. In *Proceedings of the ACM 1998 Workshop on Java for High-Performance Network Computing*; Palo Alto, California, February 1998.
21. Fox G, Furmanski W. 1996. Towards Web/Java Based High Performance Distributed Computing - an Evolving Virtual Machine. In *Proceedings of the 5th. IEEE Symposium on High Performance Distributed Computing*; 1996.
22. Gamma E, Helm R, Johnson R, Vlissides J, Booch G. *Design Patterns - Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.
23. Geist G, Sunderam V. 1992. Network-Based Concurrent Computing on the PVM System. *Concurrency: Practice and Experience* 1992; 4(4):293-311.
24. Getov V, Flynn-Hummel S, Mintchev S. 1998. High-Performance Parallel Programming in Java: Exploiting Native Libraries. In *Proceedings of the ACM 1998 Workshop on Java for High Performance Network Computing*; Palo Alto, USA, February 1998.

25. Herlihy M, Warres M. 1999. A Tale of Two Directories: Implementing Distributed Shared Objects in Java. In *Proceedings of the ACM 1999 Java Grande Conference*; Palo Alto, USA, June 1999.
26. Hoare C. 1974. Monitors: An Operating System Structuring Concept. *Communications of the ACM* 1974; 17(10):549-557.
27. Java Grande Forum. *http://www.javagrande.org.* Accessed on June 07, 2001.
28. Java Grande Forum. The Java Grande Forum Charter. *http://www.javagrande.org/jgcharter.html.* Accessed on June 07, 2001.
29. Karl H. 1998. Bridging the Gap Between Distributed Shared Memory and Message Passing. In *Proceedings of the ACM 1998 Workshop on Java for High Performance Network Computing*; Palo Alto, USA, February 1998.
30. Keleher P, Dwarkadas A, Cox A, Zwaenepoel W. 1994. TreadMarks: Distributed Shared Memory on Standard Workstations and Operating Systems. In *Proceedings of the 1994 Winter Usenix Conference*; January 1994, pp.115-131.
31. Lea D. *Concurrent Programming in Java: Design Principles and Patterns.* Addison-Wesley, 1999.
32. Loques O, Szatajnberg A, Leite J, Lobosco M. On the Integration of Configuration and Meta-Level Programming Approaches. In *Reflection and Software Engineering.* Editors: Cazzola W, Stroud R, Tisato F. Lecture Notes in Computer Science. Springer-Verlag: Heidelberg, Germany, June 2000; V. 1826, pp. 189-208.
33. Maassen J, van Nieuwpoort R, Veldema R, Bal H, Plaat A. 1999. An Efficient Implementation of Java's Remote Method Invocation. In *ACM Symposium on Principles and Practice of Parallel Programming*; Atlanta, USA, May 1999, pp. 173-182
34. Maassen J, van Nieuwpoort R, Veldema R, Bal H, Kielmann T, Jacobs C, Hofman R. 2000. Efficient Java RMI for Parallel Programming. *Technical Report*, Vrije Universiteit Amsterdam, Faculty of Sciences, March 2000. Available at http://www.cs.vu.nl/manta/. Accessed on June 07, 2001.
35. Martin P, Silva L, Silva J. 1998. A Java Interface to MPI. In *Proceeding of the 5th. European PVM/MPI Users Group Meeting*; Liverpool, UK, September 1998.
36. McCarthy J. 1960. Recursive Functions of Symbolic Expressions and Their Computation by Machine. *Communications of the ACM* 1960; 3:184-195.
37. Message Passing Interface Forum. *http://www.mpi-forum.org.* Accessed on June 07, 2001.
38. Moreira J, Midkiff S, Gupta M, Artigas P, Snir M, Lawrence R. 2000. Java Programming for High-Performance Numerical Computing. In *IBM Systems Journal*, Vol. 39, No. 1, 2000.
39. van Nieuwpoort R, Maassen J, Bal H, Kielmann T, Veldema R. 1999. Wide-Area Parallel Computing in Java. In *Proceedings of the ACM 1999 Java Grande Conference*; Palo Alto, USA, June 1999.
40. Pakin S, Karamcheti V, Chien A. 1997. Fast Messages (FM): Efficient, Portable Communication for Workstation Clusters and Massively-Parallel Processors. *IEEE Concurrency* 1997; 5:60-73.
41. Philippsen M, Zenger M. 1997. JavaParty - Transparent Remote Objects in Java. In *Concurrency: Practice and Experience* 1997; 9(11): 1225-1242.
42. Philippsen M, Haumacher M, Nester C. 2000. More Efficient Serialization and RMI for Java. *Concurrency: Practice and Experience* 2000; 12(7):495-518.
43. Sun Microsystems. 1999. The Java Hotspot^{TM} Performance Engine Architecture. *http://java.sun.com/products/hotspot/whitepaper.html.* Accessed on December 25, 2000.
44. Thurman D. jPVM: A Native Methods Interface to PVM for the Java^{TM} Platform. *http://www.chmsr.gatech.edu/jPVM/.* Accessed on June 07, 2001.
45. Transaction Processing Performance Council. *http://www.tpc.org.* Accessed on June 07, 2001.
46. Tyma P. 1998. Why are we using Java again? *Communications of the ACM* 1998;41(6): 38-41.
47. Veldema R, van Nieuwpoort R, Maassen J, Bal H, Plaat A. 1998. Efficient Remote Method Invocation. In *Proceedings of the ACM Symposium on Principles and Practice of Parallel Programming (PPoPP), July 1999.*
48. Veldema R, Bhoedjang R, Bal H. 1999. Distributed Shared Memory Management for Java. *Technical Report*, Faculty of Sciences, Vrije Universiteit, Amsterdam, the Netherlands, November 1999.
49. Wen Y, Carpenter B, Fox G, Zhang G. 1998. Java Data Parallel Extensions with Runtime System Support. In *Proceedings of the Fifth International Conference on High Performance Computing*; Madras, India, December 1998.
50. Yu W, Cox A. 1997. Java/DSM: A Platform for Heterogeneous Computing. In *Proceedings of the ACM 1997 Workshop on Java for Science and Engineering Computation*; June 1997.
51. Zhou Y, Iftode L, Li K. 1996. Performance Evaluation of Two Home-Based Lazy Release Consistency Protocols for Shared Virtual Memory Systems. In *Proceedings of the 2nd Symposium on Operating Systems Design and Implementation*, October 1996.