CONCURRENCY—PRACTICE AND EXPERIENCE

Concurrency: Pract. Exper. 2000; **00**:1–6 *Prepared using* cpeauth.cls *[Version: 2000/05/12 v2.0]*

Java for High-Performance Computing

M. Lobosco[1,*,†], C. Amorim[2,†], and O. Loques[3,‡]

[1] *Ph.D. Student, PESC - COPPE - UFRJ*

[2] *PESC - COPPE - UFRJ*

[3] *IC - UFF*

SUMMARY

There has been an increasing research interest in extending the use of Java towards high-performance demanding applications such as scalable web servers, multimedia applications, and large-scale scientific applications. However, given the low performance of current Java implementations, these application domains pose new challenges to both the application designer and systems developer. In this paper we describe and classify several important proposals and environments that tackle Java's performance bottlenecks in order to make the language an effective option for high-performance computing. We further survey most 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. We show that most of the proposed solutions can be classified according to some combination of the three basic

*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

parameters: the model adopted for inter-process communication, language extensions, and the implementation strategy. In addition, we examine other relevant issues, such as interoperability, portability, and garbage collection.

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

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 of its main objectives is to allow the portability of programs among different hardware and operating system platforms. This objective is portrayed by the known slogan "Write once, run everywhere". The approach to reach this goal was the adoption of a standardized supporting platform denominated Java Virtual Machine (JVM). The Java compiler generates a platform independent pseudo-code, denominated *bytecode*, which can then be executed in any computational environment (hardware & operating system) that supports a Java *bytecode* interpreter, included in the standard JVM.

The price paid for the portability, achieved through interpretation, as could be expected, is performance. Several efforts 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 [25]. Recent results [22] showed that optimized Java code performs comparably to Fortran for some numerically-intensive regular computations. However, these improvements were not enough to ensure that Java performs as well as

C. Nevertheless, numerous systems for high-performance computing based on the Java environment have been proposed in recent years. The target applications of these systems are those of large-scale computational nature, potentially requiring any combination of computers, networks, I/O, and memory, as defined by the Java Grande Forum [26]. Examples of such applications are 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, was not enough to reduce the great interest in its use in the development of high-performance computing environments.

Then, why to 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 are attractive enough to the development of software projects, especially those intended to 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 [6, 9, 18]. In addition, the growing popularity of the language helps to explain its use in the high-performance computing area.

In this paper we describe and classify some Java-based projects aiming directly or indirectly at supporting the development of high-performance 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 algorithms will be also discussed when appropriate.

The remainder of this paper is organized as follows. In section 2, we describe the basic support for parallel computing/programming provided by Java, as well as some other features that are relevant to understand the proposals here described. Readers that are familiar with Java's concurrency features can skip this section. In section 3, we describe the parameters that we have selected and that were used to classify the selected proposals. In section 4, we describe the Java environments and mechanisms for supporting high-performance computing that were included in this survey. Section 5 presents a classification of the systems, based on the parameters described in section 3. Section 6 concludes this work.

The Java Language

Although Java is a relatively recent language, introduced in 1992, the ideas underlying the language are not new [47]: 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 became so popular, if it did not bring anything substantially original? Two reasons seem to have contributed to its success. First, Java is a subset of an already known and widespread language, C++, incorporating multithreading, synchronization, and network communication, without relying on external libraries. Second, and perhaps the main reason, is the provision of features designed to help the development of Internet applications - the language integration to 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 parallel programming and communication, we describe these features in the next sections.

Multithreading and Synchronization

Programming with threads in Java is more immediate than with languages as C and C++. This happens because Java already provides a native parallel programming model, that includes support for multithreading in the language. The package *java.lang* provides a *Thread* class that supports methods to initiate, execute, stop 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 showed in the line 1 of the Code 1, and supplies a run method, which will be invoked when the thread 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 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 thread execution starts.

The creation of a thread follows the same pattern of object creation 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 suspending, stopping, continuing the execution of a thread, and assigning priorities to it are also available.

```
public class Mmult {

  public static void main(String args[]) {
    /* declare variables, initialize or read matrix values */ ...
    /* Process matrix, by row and column. Create a thread to process
       each element in the resulting matrix */
    num_threads = 0;
    for(row = 0; row ¡ size; row++) {
      for (column = 0; column ¡ size; column++) {
        /* set parameter p */
        threads[num_threads] = new mmultThread(p);
        threads[num_threads].start();
        num_threads++;
      }
    }
   * Print results */
  }
```

```
18 }
```

Code 2 - Another example of matrix multiplication in which a thread is created (line 12), which multiplies each row by a column of the matrix. The thread is then started (line 13).

Besides multithreading, the language also includes a set of synchronization primitives. Those primitives are based on an adaptation of the classic monitor paradigm proposed in [24]. The standard semantics of Java allows the methods of a class to execute concurrently. The *synchronized* reserved word can be associated to given methods in order to specify that they cannot execute concurrently. Then, these methods can only execute in a mutual-exclusion fashion according to the monitor paradigm. The example (see code 3), extracted 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 was 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.

```
class Barrier {

  protected final int parties;
  protected int count; // parties currently being waited for
  protected int resets = 0; // times barrier has been tripped

  Barrier(int c) { count = parties = c; }

```

```
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: it is guaranteed with the use of *synchronized* reserved word (line 09).

As has been identified by the Application and Concurrency Work Group of the Java Grande Forum [27], thread synchronization introduces a potential performance bottleneck (see also next sub-section), which ultimately prevent Java applications with large number of threads to scale.

Prepared using cpeauth.cls

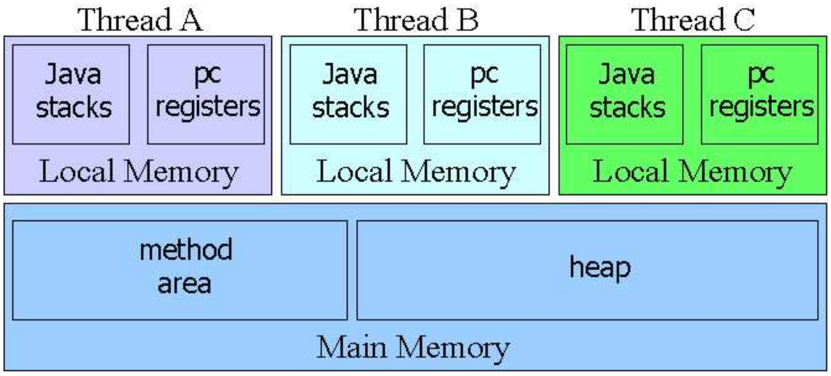

Figure 1. The internal architecture of the Java Virtual Machine's Memory

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 specific 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

Prepared using cpeauth.cls

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.

Communication

Java offers a rich set of tools and APIs 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.

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 to define 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.

Prepared using cpeauth.cls

In order to use RMI, the programmer must structure his/her application obeying the client/server paradigm, whereby a remote object represents the server and the client corresponds to the object that invokes the method. In addition, a simple programming recipe that includes inheriting a special *Remote* class and using some standard methods in the application code must be followed. A standard Java tool, *rmic*, is used to automatically generate a stub (auxiliary code), which works 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 RPC implementations, the skeleton figure performs this role. For a method invocation, the stub establishes the connection with the remote JVM, marshals the invocation parameters, waits for the method invocation to complete, unmarshals all results or exceptions, and returns the outcome to the invoker.

The arguments sent to or the values returned from a remote object can be any serializable object. This includes primitive types, remote objects, and non-remote Java objects that implement the *java.io.Serializable* interface. The RMI system dynamically loads classes, associated to parameters, or return values, which are not available locally. Parameters are always passed by reference if they refer to remote objects; otherwise they are passed by copy.

In case of objects passed by copy, it is necessary to execute a serialization operation that transforms objects into arrays of bytes, including all instance variables of primitive and non-primitive types declared by the objects. For non-primitive types, the complete reference graph is serialized, even if it is cyclic.

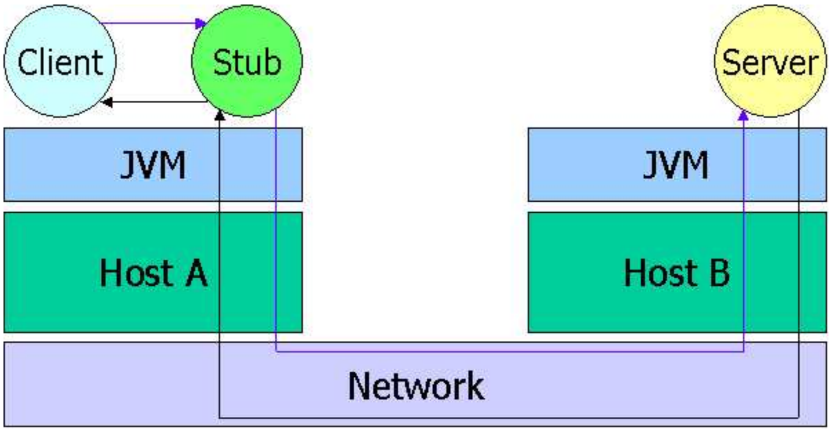

Figure 2. The RMI protocol

The RMI definition enables the target JVM to know in advance the argument types of the JVM calls. Given that the RMI supports polymorphism† , the type of each argument can be any subtype of the arguments declared by the method. Thus, the byte array representation must also incorporate the information about the serialized types. Therefore, any RMI that passes objects by value must also declare their types. Note that this time-consuming process is unnecessary for many scientific applications and may degrade the application performance.

Programmers are free to write their own marshaling and unmarshaling routines, which will be invoked by the serialization mechanism. However, programmers often prefer to use the serialization methods automatically generated by the compiler. These methods use a structural

†The use of any superclass of the subclass to refer to the instance of the subtype.

reflection[‡] mechanism, which provides the appropriate byte array representation and finds dynamically the type of each object. In spite of simplifying the programmer's task and offering greater flexibility in programming, this model introduces a large overhead due to the great number of operations that have to be executed dynamically, which can limit its use in high-performance applications.

The Java memory model 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 have been not referenced any longer.

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 [50] and Fast Messages [41]. 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 configure the communication protocol most suitable for a given application [33].

Also, it would be useful the addition of collective communication in the language, such as scatter and gather, all gather, and all-to-all. Java already has the multicast collective

[‡]Structural Reflection can be defined as the "ability of a language to provide a complete reification of both the program currently executed as well as a complete reification of its abstract data types".

communication implemented via sockets, but an implementation of this pattern in the context of a high-performance communication support would be useful.

Classification parameters

Three basic issues are relevant to the design and implementation of a Java environment for high-performance computing: (a) the model adopted for inter-process communication; (b) modifications introduced to the languages' semantics and syntax; and (c) implementation strategy. Explicitly or implicitly they appear as distinguishing features in the surveyed proposals. Such features allow us to consistently divide the works we surveyed, producing a clear classification of alternatives that have been explored so far.

The way in which processes communicate is an important issue for implementing an effective environment for high-performance computing. Three 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) [20] and the Message Passing Interface (MPI) [39], 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 [53] 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 achieved performance are determined by the mechanism adopted for inter-process communication.

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

The last issue is related to the strategy adopted for the environment implementation, which affects code portability and environment performance. This survey has identified five main approaches: (a) the use of a pre-compiler; (b) modification of the Java compiler; (c) modification of the JVM; (d) extensions based on libraries written in Java; and (e) the use of native functions of a particular environment.

Further issues, including garbage collection, interoperability with other Java virtual machines, and portability are also important. The first is particularly important, since the language specification assumes the existence of an automatic storage management system; this garbage collection mechanism has to work transparently for local and remote objects. For example, the RMI mechanism has to garbage collect remote objects that have been not referenced any longer. 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 and portability are desirable features not directly dependent of the basic language design. However, these three 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.

Environments

In this section we describe several proposals aiming at transforming Java into an efficient environment for high-performance computing. Some of them tackle specific performance bottlenecks, such as the high costs associated to 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 a technique in their implementations, we consider the most significant one for classification purposes. Nevertheless, in section 5 we make a crossover comparison taking into account all the techniques used in each of the proposals discussed before.

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

Java's Semantics / Syntax Unmodified

Changes to the JVM

This section presents systems that provide the shared-memory abstraction, through changes made to the internals of the basic JVM, transparently, i.e., without modifying either the semantics or the syntax of Java. Two systems fall on this class: MultiJav [12], and cJVM [1, 2, 3].

MultiJav

One of the main objectives of MultiJav is to maintain the portability of the Java language, allowing its use in heterogeneous hardware platforms. The MultiJav's approach is to implement itself the distributed shared-memory (DSM) model into Java by modifying the JVM, while using Java constructs to support concurrency, thus avoiding changing the language definition. Sharing is object-based in MultiJav. An apparent shortcoming of MultiJav is that all the objects are potentially shared, since the programmer cannot declare which objects are to be shared. However, the MultiJav runtime system through an analysis of the load/store instructions of the *bytecode* being executed can detect automatically which objects should be shared, catering for their consistent usage. This technique seems to be the main contribution of the work. Different threads are permitted to access variables of a 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.

A MultiJav program begins execution in one virtual machine, named root, but spawned threads can migrate to another machine, afterwards. In order to attain to the standard Java semantics, monitors are global to all the sites that participate in a computation. Thus, each participating site contains queues with local threads and requesting threads, which represent the threads of higher priority in remote sites that requested access to the monitor. 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 efforts required for such a task. Also, the overheads that the adopted synchronization management mechanism introduces are unclear. Unfortunately, a performance analysis is not available for MultiJav, perhaps because has not been implemented yet.

cJVM

cJVM [1, 2, 3] has been developed at IBM Haifa Research Laboratory in Israel. cJVM supports the idea of single system image (SSI) in which a collection of processes can execute in a distributed fashion with each process running on a different node. To implement the SSI illusion, cJVM uses the *proxy* design pattern [19].

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.

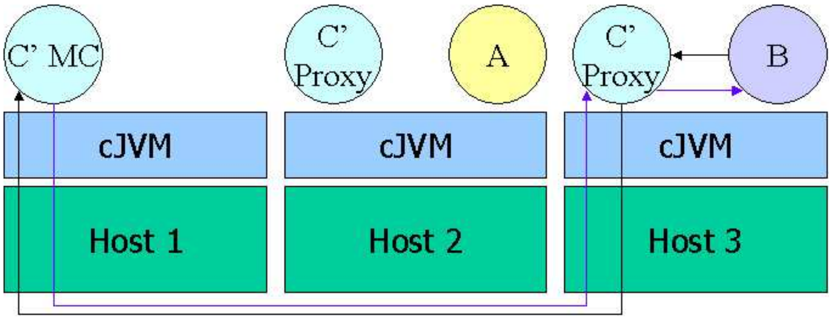

Figure 3. The master copy (MC) of object C was created at host 1 since it was the first node to make reference to this object. Object B accesses object C through C's proxy. The proxy then forwards the method invocation to the MC of the object C.

Aiming at performance optimization, during class loading, its associate 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: (a) standard proxy which transfers all the operations to the master copy; (b) read-only proxy which applies the operations locally, based on the fact that it is guaranteed to access only fields that never change, so the proxy can replicate and maintain these fields; and (c) proxy that locally invokes methods without state, since these are methods that do not access object fields.

The introduction of proxies and redirection of methods make the execution stack to be distributed among multiple threads across 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 semantics 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 best node 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 inspired on the TPC-C benchmarks [46] and consists of N warehouses 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 on considering that the application is highly parallel a linear speed-up or so 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 [26].

Modification of Java's Semantics / Syntax

Changes to the JVM

Java/DSM

Java/DSM [52] 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 software DSM library, which enables the JVM to be extended across the network.

Java/DSM is similar to the systems presented in the last subsection (e.g., MultiJav), except for the changes to Java's semantics. In contrast to those systems, the heap is allocated in the shared memory area as shown in Figure 4, which is created with the use of TreadMarks, and classes read by the JVM also are allocated automatically in the shared memory. Two restrictions are imposed to the programmer: (a) a thread cannot migrate between machines; and (b) thread's location is not transparent. The first restriction hinders dynamic load balancing activities, preventing thread's migration from overloaded processors to idle ones, whereas the second restriction requires the programmer to be aware of thread's location.

Java/DSM extends the Boehm and Weiser collector [8], which is a distinguishing contribution of the work. The garbage collector of each machine maintains two lists; one containing remote references for objects created locally (export list), and other keeping references to remote objects (import list). The lists contain an estimate of the actual cross-machine reference set, which are used only for garbage collection purposes. Before a message is sent to other machine, the runtime DSM support invokes the garbage collector that inspects the message contents, in order to verify if it contains valid references to local objects and references that are identified are inserted in the export list. Likewise, incoming messages are inspected and references to remote objects are collected and inserted in the import list. Garbage collection is performed

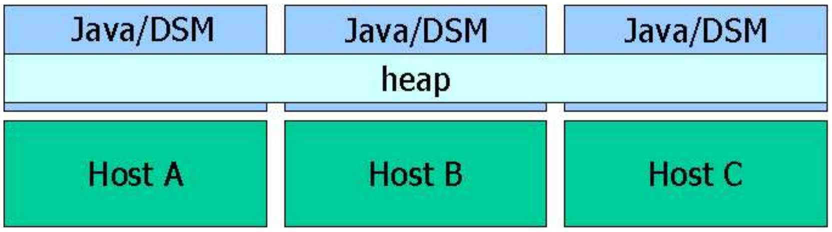

Figure 4. The heap in Java/DSM is shared among all Java/DSM nodes.

using a weighted reference counting algorithm to decide when a reference can be discarded of the export list. For most of the time, each machine independently executes the garbage collection, although some synchronization operations are required once 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 from which the conversion should be done. In order to perform object identification efficiently, Java/DSM requires that only objects of the same size can 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 the Java standard object representation includes only two components: the handle and the body. The handle contains a pointer to the structure that stores type information of all 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 know the size of the objects the page contains. Once it is found the beginning of the object to which the address belongs to, the back pointer can be followed to discover 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 preliminary comparison between Java/DSM and standard Java RMI was reported in [52] 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 [49], from Vrije University in The Netherlands, implements a software DSM abstraction on a cluster through a special run time support and an associate compiler. The compiler, which generates native code, is also used to make optimizations, such as data prefetching. Jackal required some semantics change to Java, however.

The Jackal runtime system implements a cache coherence protocol for the memory units, called regions, which are defined as either objects or fixed-size array partitions. The coherence protocol is based on self-invalidation, which every time a thread reaches a synchronization point it invalidates its own data, ensuring the coherence of subsequent accesses. Although such protocol is simple it can invalidate data that will not be touched by any other node, thus adding unnecessary overheads to the coherence mechanism. To implement the self-invalidation protocol, each thread maintains a control list of the areas 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 area 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, requests a copy of the area, and stores the received copy in the same address location it is kept in the home node.

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 (GC) cannot discard objects that are referenced by other objects in remote nodes. When the amount of this kind of 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 the global GC is due to the amount of communication and synchronization among the involved nodes.

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.

A micro benchmark was executed to measure the overhead of garbage collection, including access validation plus latency and throughput of object transfers. All tests were run on a cluster of 200MHz Pentium-Pros, running Linux, connected by a Myrinet network, using LFC [7] 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 Mbyte/s to 24 Mbyte/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 and in Ray-tracing, both the runtime system and garbage collection generated a large overhead. The results from the Web server were not made available.

Using 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 in this category: Charlotte [28] and Aleph [23], from Brown University.

Charlotte

Programs in Charlotte 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. The access to the shared data is made through special methods: *get()* and *set()*. In Charlotte, sharing is object-based. In a read access, if it is detected that an item is invalid then a new copy is requested to the object manager. In a write access, the object is marked as updated so that all of the modified objects are 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 read overhead, every read operation retrieves a group of objects from the manager instead of one. It is up to the programmer, with the use of appropriate annotations, to give Charlotte some information of which data will be really used by a routine. This procedure is also adopted for write operations. The annotations for accessing groups of data resemble read / write operations used in the message-passing approach; the difference is that the data does not need to be explicitly read / write using primitives like send/receive. Charlotte allows the verification of annotations at runtime. In this case, even if the programmer makes incorrect annotations in the code, the program may continue to work correctly. The overhead of this verification can be transferred to the compiler. The use of annotations may be effective on improving performance of an application and it is the main contribution of the proposal. However, to take advantage of annotations, a good knowledge of the application is required from the programmer. Moreover, it is unclear if irregular applications could get a significant benefit from such an approach.

The Charlotte author suggested additional optimizations: since the manager knows the data that 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 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.

In [28] execution times for a matrix multiplication application are presented, and results for several versions of Charlotte are compared with a version of the code for the same application based on message passing. The results indicate that, for the version of Charlotte that does not verify annotations during execution time, assuming that the compiler could do it, yields execution times that are competitive with 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 PE's 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

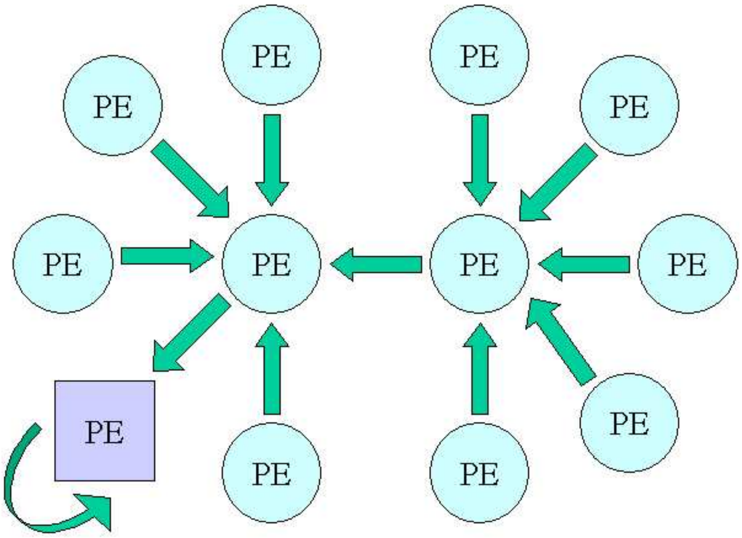

Figure 5. The Arrow Protocol. Each processor element (PE) uses a pointer to indicate the path that must be followed to reach a given object. If a PE points to itself, either the object is located within the PE or it will be shortly moved to it (represented by the square in the picture). If the link points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

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, if the link points to another PE, then the object belongs to a component of the tree. Informally, it can be said that, if the PE is not owner of an object, 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 [16] 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 provide then a *run()* method, which will be called at the receiver upon message arrival.

Results are presented for the three directory-based protocols on 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 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 limit the analysis. The use of a library to implement the environment may suggest poor performance; especially when compared to those implementations that either modify the JVM or rely on compiler support.

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 gives 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 work.

MultiJav keeps unchanged Java's semantics and syntax, despite of introducing the distributed shared memory model by modifying the JVM. Such an approach offers two advantages: (a) the potential of reusing standard JVM code, and (b) application performance is better than using Java library. MultiJav offers also an automatic mechanism for detecting shared data. The disadvantages of MultiJav are (a) lack of interoperability with others implementations, and (b) less portability when compared to library implementations.

In comparison with MultiJav, Java/DSM has two disadvantages: (a) Thread's location is not transparent; and (b) threads cannot migrate, which prevents dynamic load balancing. Unfortunately, both Java/DSM and MultiJav have not presented any performance results, so far.

cJVM is similar to MultiJav since that both keep unchanged the Java's semantics / syntax and extend the JVM to support DSM. However, they differ in the way that DSM is implemented. cJVM prefers the proxy design pattern (PDP) [19] to implement the Single System Image abstraction. In the PDP model, there is just one instance of object (called master) for all 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. MultiJav allows multiple copies of instances of objects to coexist, uses a diffing mechanism to detect changes made to the objects at execution time, and disseminates updates at synchronization points. Java/DSM uses the TreadMarks software DSM library that implements also 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, causing overheads near 14%. Another source of significant overhead in Jackal is the use of self-invalidation protocol in which data are invalidated even if data have not been modified by any other node. Also, Jackal's new memory model is also arguable.

By implementing DSM using Java library, Charlotte and Aleph have some pros and cons. For instance, both favor program portability over performance loss when compared with MultiJav, cJVM, or Java/DSM. In addition, Charlotte offers to 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. Overall, both works reported few performance results, thus it is important that more experiments be carried out before any conclusion can be withdraw from those proposed environments. Indeed, Challote might investigate whether regular applications can benefit or not from annotations, and Aleph's authors might investigate how well the arrow protocol and its hybrid version perform across several classes of applications.

Inter- process Communication using the Message Passing Model

Java's Semantics / Syntax Unmodified

Using Native Library

mpiJava

The mpiJava [5], from NPAC at Syracuse University, is a Java interface for existing MPI [39] 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 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 in Java.

Some MPI functions omitted the argument that indicates the array size, since this can be obtained through the Java's length method. The MPI destructor function is called by the Java's finalize method, except for Comm and Request, which have explicit *Free* members. The introduction of explicit calls to a method that releases memory breaks up the 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, and therefore the performance difference can be attributed mostly to the JVM. The results were obtained using both: (a) WMPI, a Windows-

[§]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.

Prepared using *cpeauth.cls*

based implementation of MPI, running on two dual processor (P6 200MHz) Windows NT 4 workstations with 128 Mbytes of DRAM, and (b) a Solaris version of MPICH, running on two dual processor (200MHz Ultrasparc) Solaris workstations with 256Mbytes of DRAM. The mpiJava overhead under WMPI was about 100 ms whereas the MPICH overhead was between 250 and 300 μs, that is significantly lower.

Java-to-C Interface Generator (JCI)

The Java-to-C Interface Generator (JCI) [21] has been developed at IBM T. J. Watson Research Center. JCI is a Java interface generator to C 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 restrictions of mpiJava are not found in JCI, or they can be eliminated using some 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's 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 order to pass array blocks, with blocks beginning at certain indexes, as parameters in function calls, which is not possible under Java, JCI creates the method *JCI.section (array, index)*.

A performance comparison was carried out between C and Fortran-77 code and Java linked with native libraries using two benchmarks: IS, from the NAS suite, written in C, and MATMUL, from the PARKBENCH suite, written in Fortran-77. The IS benchmark runs on two different platforms: (a) a Fujitsu AP3000 (Ultrasparc 167 MHz nodes); and (b) 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 Spark workstation cluster and on an IBM SP2 system (66 MHz Power2 "thin1" nodes). The results showed that Fortran-based MATMUL outperforms Java from 5% to 10% whereas Java -based IS were twice slower than C versions. The explanation is that in MATMUL most of the performance-sensitive calculations were performed by the native code.

A lot of works describe Java binds either to MPI or to PVM, and can also be placed in current section (interprocess communication using message passing, Java's syntax / semantics unmodified, use of native library). JPVM [17] presents a simple front-end to PVM. JPVM [45] is an interface that was developed using features of native methods and allows Java applications to use PVM. MPIJ [15] is a Java-based implementation of MPI integrated with DOGMA (Distributed Object Group Metacomputing Architecture). JMPI [14] is an MPI environment built on top of JPVM [42]. JavaWMPI [37] is a MPI version built on MPI for Windows.

Changes to Java's Semantics / Syntax Changes

Changing the Compiler

Manta

Manta [48, 35, 40] 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 three main sources of Java's overheads: 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 serialization/un-serialization codes must be generated at compile time, thus reducing the overheads of dynamic inspection. Manta's protocol for serialization has also some scope for optimization. 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 *hashtable* is also used to keep serialized objects (the *hashtable* is created only if parameters in remote calls are objects). Whenever a replicated object is found

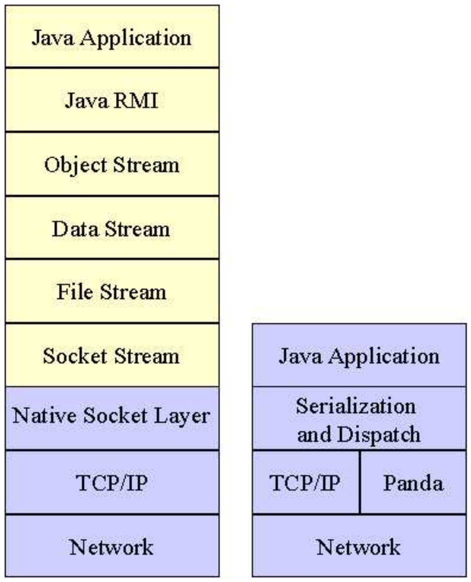

Figure 6. The structures of Sun (leftmost) and Manta RMI protocol. Layers in dark denote compiled code.

the *hashtable* is accessed. 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 better 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 for Panda, a user's level communication library that has independent interfaces to both the hardware and network protocol. For the Manta tests using the Myrinet switch, Panda was implemented on top of LFC [7].

In Manta, the computation nodes use an optimized RMI protocol for communication that might cause interoperability problems with other JVMs since Manta has modified the original Java protocol. Furthermore, RMI methods can be polymorphic, so Manta must be able to receive and send *bytecodes* in order to interoperate with other Java virtual machines. Given that the Manta compiler generates native code, using the optimized protocol among Manta machines and the standard RMI between Manta and other Java virtual machines can solve the interoperability problem. The Manta' solution to the polymorphism problem is to let the compiler generate both *bytecodes* and native code. The former is placed in a HTTP server, allowing remote JVMs to access them. In case of receiving *bytecodes* from a remote JVM, the received *bytecode* is dynamically compiled to object code, and soon after the object code is linked to the application, using the operating system's dynamic linking interface *dlopen()*.

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 [38]. 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 [13] for remote objects.

The performance impact of all the above optimizations appears when Manta is compared with JDK. For the simplest remote call without any input parameters or result, the Manta latency is 35.2 μs[¶] against 1210.14 μs[‖] and 1719.87 μs^5 for JDK w/o JIT, respectively. The JDK versions 1.1.3 and 1.1.6 were used when running without JIT and with JIT, respectively. This result is overestimated because Manta and JDK used different switches (Myrinet in Manta x Fast Ethernet in JDK) with significant difference in both latency and effective bandwidth delivered to the application.

Manta alsoshows performance results for TSP (Traveling Sales Person), SOR (Sucessive Over Relaxation) and IDA* (Iterative Deepening A*) benchmarks, which were widely favorable to Manta when compared with JDK performance. In [40] are also reported uses of Manta in metacomputing applications.

Manta modified the syntax of Java by introducing the reserved word *remote*, which permits the programmer indicate which classes can be remotely invoked, and replace the language standard mechanism that requires the inheritance of the class *java.rmi.server.Unicast.RemoteObject*. This new operator provides support to the creation of objects in remote machines. These syntax modifications prevent Manta from reusing code written for other Java machines.

Other characteristics limit the use of Manta, 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

[¶]Running on a 200MHz Pentium Pro, 128Mb of memory, connected by 1.2Gbit/sec Myrinet running BSD/OS 3.0.

[‖]Running on 300MHz Ultrasparc running Solaris 2.5.1 connected by 100Mbit/s Fast Ethernet.

should start at the same time. Furthermore, Manta does not support the heterogeneity and safety of the Java model.

Using a Pre-compiler

JDPE (Java Dates Parallel Extensions) [51] and JavaParty [42] propose semantics and syntax changes to Java and developed a pre-compiler to support the changes they introduced. The message passing approach is used to enable inter-process communication.

JDPE

JDPE focus on the potential benefits of including some characteristics of High-Performance Fortran, such as the distributed array model, array intrinsic functions, and libraries, which could make Java an attractive language for programming in the SPMD model.

JDPE provides distributed arrays as language primitive and distributed control constructs to facilitate access to the elements of a local array. Under this model, programmers need not to know the physical location of any particular array element. To do so, JDPE 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; *Location*, which is an abstract element of *Range*, hence *Range* can be considered as a group of *Locations*. In addition, there still exist the new two classes: *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 a particular subset of a distributed array. JDPE's active process construct defines the group of processes that share an active thread of control which can be explicitly established through the control operator *on*, for example, *on (p)* {...}. In the construct body, the group of active processes is changed to the p group, and all the operations on the distributed arrays are executed on this active p group. The *at* operator is similar to *on*, except that its body is executed only in the processor that is the owner of 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 in the processor that owns the location 13 of the array x. The *over* operator implements a distributed parallel loop.

Collective communication libraries are supplied to the JPDE programmer to ease his task of controlling the data movement 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 amount of elements of a given dimension either in a cyclic or off-edge way. The *writeHalo* command supports *ghost* regions in the communication, allowing reducing the amount of memory copies and communication traffic during collective communication. Communication is deadlock free and new communication libraries can be integrated to the supplied library. The JDPE environment comprises a pre-compiler and runtime libraries.

Cholesky and Jacobi applications were used for performance evaluation purposes, but the performance results reported are unclear.

JavaParty

JavaParty [42] 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 alteration to the language was the introduction of a new reserved word, called remote. Remote 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 mechanisms between objects. The run time system and compiler are responsible for this work, as well as, for dealing with of 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 could be also modified at run time.

The run time system implements load balancing, network partition, and monitoring of interactions between objects. In this way, the run time system, or even the programmer, can migrate objects to increase their locality. This issue is completely ignored by the Java execution environment. Indeed, if one of two objects invokes a method at the other using

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

RMI, Java activates the RMI mechanism even if they share the same machine. In contrast, JavaParty will check the object location and a local method invocation will be made. The local method invocation takes 0.7μs against 2.8ms of a RMI invocation, thus the penalty for using RMI unnecessarily is very high, up to 4000 times slower.

Another alteration introduced to the language is the permission for methods and static variables to be remotely accessed, which is not allowed with 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 of the class originally declared by the programmer.

The JavaParty work reports no performance results and gives examples to illustrate only how the compiler generates code for static methods and variables.

Using a Java Library

Java//

Java// [10], 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 [19] in order to facilitate its implementation.

In respect to the object vision, both Java// and Java support passive objects but only Java// supports active objects (threads and actors), thus changing the semantics of the standard Java object. An active object is composed of an object as declared by the programmer, and a body, that is the associated object that owns the queue of pending method invocations for an object. Java// offers three mechanisms to declare active objects: (a) calls to the *Java//.newActive* method, which extended the *new* functionality to allow the declaration of active objects; (b) calls to *Java//.turnActive* which transforms a passive object already declared into an active object. Note that both *Java//.newActive* and *Java//.turnActive* give to the programmer the option of creation of the object in a remote node. The third mechanism implements the *Active* interface.

To implement active objects, the designers of Java// created a new mechanism based on two components: (a) a request queue for each object, where pending invocations can be stored; and (b) a thread for managing the queue. The object, which is the owner of a request queue, is denominated *Body*. Pending requests are executed asynchronously, and the execution order depends on the selected synchronization policy. The body object follows the FIFO behavior if the programmer provides no policy.

[††]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".

The concept of future object is applied for inter-object synchronization. A future object is simply an object created to make possible an immediate return from a method invocation. In this way, the thread that made the invocation can continue its execution as long as it needs not to invoke methods in the returned object, otherwise 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 it is data-driven synchronized. 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 which indicates whether the object *obj* is awaited or not. The latter method allows a thread to carry out any useful task instead of keep 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 provided, by managing explicitly the request queue through the methods *serveOldest()*, *serveOldest(method met)*, *serveOldestBu (method met),* and *waitARequest()* implemented by the *Body*. For implicit synchronization the method *forbid (method, condition)* is used, which works as a guard, impeding the access to the method *method* when the condition *condition* is true.

For illustration purpose, two applications, namely matrix multiplication and a collaborative application that uses a Ray-trace algorithm, were described though performance results were not presented.

UKA-Serialization and KaRMI

Serialization and Java RMI are two main sources of overheads that UKA-Serialization and KaRMI [43] 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 in four different fronts: type coding, internal buffering, buffer accessibility, and maintaining type information upon *hashtable* reset.

The type-coding problem happens because Java has to keep enough information on persistent objects stored in disks so that the objects can be retrieved afterwards, 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 object's 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 *hashtable* in such a way that objects that are re-transmitted will hold their new states. For implementing this property, two alternatives can be used, either creating a new serialization object for each method invocation or calling the *reset* method in the serialization object. The effect of both is to clean the information of all objects that were previously transmitted, including type information. The UKA-Serialization thus creates a new *reset* method, which cleans only the *hashtable*, maintaining the type information unaffected.

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 no buffering strategy is implemented in the receiver, thus it ignores the number of bytes required to perform marshaling operations to objects. The authors 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 causes 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.

KaRMI tries to improve performance of the Java RMI through several optimizations. For instance, a clean interface between RMI layers was elaborated which offers two advantages: (a) the RMI invocation requires just two additional invocations; and (b) efficient implementation of the transport layer could be made. 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 more five native calls for each remote invocation.

The results show that UKA-Serialization reduces object serialization time from 76% up to 96%, when compared with JDK serialization. For KaRMI, three classes of benchmarks were used: (a) kernels that test RMI calls between two nodes; (b) kernels that test the overload of the server from calls issued by several clients; and (c) specific applications, such as the Hamming problem, Paraffin Generation, and SOR. The benchmarks were executed on two different hardware platforms: (a) two PC Pentium II 350Mz, running Windows NT 4.0 Workstation, isolated from the LAN, and connected to each other by Ethernet, JDK1.2 (JIT enabled); and (b) a cluster of 8 DECs Alpha 500MHz, running Digital UNIX, connected by Fast Ethernet, JDK1.1.6 (regular JIT). For small size arrays, the results show that KaRMI outperforms RMI from 41% up to 84%. However, for large size arrays (e.g., 5000 elements), their performance is equivalent. Unfortunately, the authors do not compare their results with similar works 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 Architecture Interface) is an emerging industry standard developed at Cornell University for user-level network interface and Javia [11] 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 to an environment for high-performance computing. Javia consists of a group of Java classes that implement an interface with 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, denominated 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 either the JNI native interface or similar. 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 (version A) switch.

The second level, Javia-II, permits the programmer to manage directly the communication buffers, so that application's specific information can be exploited to implement better buffering policy. The management is made using the *viBuffer* class and its methods, which provide only 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 fashion similar to the Java's primitive arrays, allowing Java applications to directly transmit and receive arrays and eliminating the need of 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 authors assume 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 program that allocates memory indefinitely is potentially incorrect whereas a Javia program that does not de-allocate the buffers thus leading to a situation of insufficient memory could be correct if Java's memory management semantics is followed. 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 same platform described above.

Summary

mpiJava, JCI, Manta, JDPE, 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 syntax and semantics of several MPI functions. JCI is an interface generator from Java to C that allows programmers to benefit from existing native libraries such as MPI; besides JCI has fewer restrictions than mpiJava. In addition, preliminary performance results suggest that JCI outperforms mpiJava, however, this is only a partial conclusion as the results are not directly comparable given that they were obtained using different platforms and benchmarks.

JPDE introduced the SPMD model into Java. JDPE provides classes to work with distributed arrays and distributed control constructs that enable each process to execute a particular array subset. JDPE also provides collective communication libraries to control the data movements. Some of the collective communication library 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, network partition, and monitoring of interactions between objects. In this way JavaParty is similar to cJVM, since both give to the programmer the option to control load balancing in the system. Another interesting characteristic of JavaParty is the permission for methods and static variables to be remotely accessed, which is not allowed within traditional RMI. Regrettably, JavaParty does not present any performance result.

Java// introduces three new concepts to the language: (a) active object; (b) future object; and (c) new methods for intra-object and inter-object synchronization, which facilitate object synchronization. The concepts behind future objects and proposed synchronization methods are powerful enough to deserve inclusion into the language. On the other hand, Java// did not present any performance evaluation.

Manta translates Java directly to executable code and tackles three main sources of Java's overhead: (a) serialization; (b) RMI streams and dispatch; and (c) network protocol. The performance comparison between Manta with the original JDK is mostly favorable to Manta.

Moreover, Manta should also outperform all the other systems presented in this work. Despite of modifying the RMI protocol, Manta can interoperate with other Java Virtual Machines. However, Manta has some disadvantages: (a) some Java's characteristics are omitted in order to optimize Manta's implementation; (b) all the processes that participate in computations should start at the same time; and (c) some Java features such as portability and robustness that make it attractive for developing large and complex systems are not presented in Manta.

UKA-Serialization and KaRMI cope with serialization and RMI, which Manta identifies as two main sources of Java's overheads. Nevertheless, UKA-Serialization and KaRMI use a different implementation approach in that no compiler support is required and all the code is written in Java. In this way, the code becomes portable and can be used in other platforms.

Javia is another attempt to tackle the low performance of Java communication. Javia performance was excellent just 1% slower than C for the benchmarks tested. However, the programmer must manage the memory allocated for communication, thus violating the automatic memory management of Java.

Classification

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

Works in the semantics & syntactic column refer to the language changes that can be seen by the programmer. Note that our classification of cJVM considers the fact that the change in the *new* opcode is transparent to the programmer.

As described in this paper, the majority of the proposals modify the semantics and/or the syntax of the language. This happens because: (a) they add new capabilities to Java, such as Aleph's declaration of global objects; (b) They try to ease the programmer's task, such as Manta's and JavaParty's *remote* keyword; or (c) some project decision has forced the semantics / syntax to change, such as Javia's need of an explicit buffer de-allocation by the programmer. Only four of the surveyed systems do not change Javas semantics & 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 the JCI, which is a JNI-like mechanism usually familiar to the programmer.

Most of the systems use the message-passing model for inter-process communication, since this model is more widespread than the shared-memory one for high-performance computing. However, in a Java context, it can be argued that the latter is more natural because the thread model, which assumes a memory shared among all the threads in the Java Virtual Machine, already expresses it. Besides, existing multithreaded programs can potentially run without modifications on a distributed shared-memory environment. Nevertheless, a potential disadvantage of DSM is that often this model performs worse than the message-passing model.

Some systems we 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 primary 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 socket and RMI, are also available. However, in our classification we were restricted to the main inter-process communication the author focused.

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, the interoperability is potentially unfeasible. Indeed, 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, Table 1 shows that the systems that do not provide interoperability 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 case, garbage collection is also related to the approach chosen for the environment implementation. Again, if the proposal is implemented through a 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, probably 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, we can see that 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.

Concluding Remarks

In this paper we have described and classified recent Java-based proposals for high-performance computing. Our classification scheme has taken into account some critical parameters, such as 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 consider 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, portability, and garbage collection algorithms have been also treated.

The majority of the proposed systems described in this work has 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 developing of a Java-based system for high performance computing should not incur in the modification of the characteristics of the language that turned it 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 could 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 language could also be interesting, such as the addition of a message passing library following MPI or PVM styles. Finally, we suggest that one can take advantage of the extra information that is available for free in the byte code, and in the virtual machine interpreter state, in order to optimize the execution of parallel applications.

We have also pointed out some potential Java issues that can affect the performance of high-performance applications. More specifically, (a) the performance of thread synchronization primitives; (b) the data serialization operation for communication, which determines at run-time the type of the parameters to be transmitted, therefore adding 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-pointer and array bounds performed by the JVM; (e) 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, also preventing optimizations [22].

A general concern is the lack of performance measures related to the vast majority of the proposals. This might be explained due to the fact that the systems are mostly either basic proposals or started the implementation recently. Unfortunately, even systems which are in a relatively advanced implementation stage, and have reported some potentially interesting experimental results, tend to use their own benchmarks in an ad-hoc fashion. This fact makes difficult more meaningful comparison between their findings.

Therefore, we expect that future works in this research area 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 [26]. This benchmark is divided in three sections: (a) low-level operations; (b) kernels of application, such as FFT and SOR; and (c) large scale applications, such as Ray-tracer and Monte Carlo simulation. Additional benchmarks to measure the associate 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's working memory and the main memory as well as the contention during accesses to the main memory. Furthermore, it would be valuable the addition of Internet related kernels and applications to the benchmark, since Java has been broadly used in the Internet.

Table I. Classification

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

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 (ICPP 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 (ICS)*; 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, 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, California, USA, February 1998.
7. 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.
8. Boehm H, Weiser M. 1988. Garbage Collection in an Uncooperative Environment. *Software: Practice and Experience* 1988; 18(9):807-820.
9. 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.
10. Caromel D, Klauser W, Vayssire J. 1998. Towards Seamless Computing and Metacomputing in Java. *Concurrency: Practice and Experience* 1998; 10(11-13):1043-106.
11. Chang C, von Eicken T. 1999. Interfacing Java to the Virtual Inteface Architecture. In *Proceedings of the ACM 1999 Java Grande Conference*; Palo Alto, California, USA, June 1999.
12. 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 (PDPTA'98); Las Vegas, Nevada, USA, July 1998.

13. Collins G. 1960. A method for overlapping and erasure of lists. *Communications of the ACM* 1960, 3(12):655-657.
14. Dincer K, Ozbas K. 1998. jmpi and a Performance Instrumentation Analysis and Visualization Toll for jmpi. In *Proceedings of the First UK Workshop on Java for High Performance Network Computing*; Southampton, UK, September 1998.
15. DOGMA. *http://www.zodiac.cs.byu.edu/DOGMA/DOGMA.html*. Accessed on December, 25.
16. 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 (ISCA'92)*; May 1992.
17. 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.
18. 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.
19. Gamma E, Helm R, Johnson R, Vlissides J, Booch G. *Design Patterns - Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.
20. Geist G, Sunderam V. 1992. Network-based concurrent computing on the PVM system. *Concurrency: Practice and Experience* 1992; 4(4):293-311.
21. 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, California, USA, February 1998.
22. Gupta M, Midkiff S, Moreira J. 2000. High Performance Numerical Computing in Java: Compilar, Language and Application Solutions. *Tutorial at the ACM / IEEE Supercomputing*; Dallas, USA, November 2000.
23. 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, California, USA, June 1999.
24. Hoare C. 1974. Monitors: An Operating System Structuring Concept. *Communications of the ACM* 1974; 17(10):549-557.

25. Sun Microsystems. 1999. The Java Hotspot^{TM} Performance Engine Architecture. *http://java.sun.com/products/hotspot/whitepaper.html.* Accessed on December, 25.

26. Java Grande Forum. *http://www.javagrande.org.* Accessed on December, 25.

27. Java Grande Forum. The Java Grande Forum Charter. *http://www.javagrande.org/Charter.html.* Accessed on December, 25.

28. 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, California, USA, February 1998.

29. Keleher P, Cox A, Zwaenepoel W. 1992. Lazy release consistency for software distributed shared memory. In *Proceedings of the Nineteenth International Symposium on Computer Architecture*; May 1992, pp. 13-21.

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. Doug L. *Concurrent Programming in Java: Design Principles and Patterns*, second edition. Addison-Wesley, 1999.

32. Lindholm T, Yellin F. *The Java Virtual Machine Specification*, second edition. Addison-Wesley, 1999.

33. 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. 191-210.

34. Wang M, Lau F, Xu Z. 1999. JESSICA: Java-Enabled Single System Image Computing Architecture. In *Proceedings of the 1999 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA99)*; June, 1999.

35. Maassen J, van Nieuwpoort R, Veldema R, Bal H, Plaat A. 1999. An Efficient Implementation of Java's Remote Method Invocation. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*; Atlanta, GA, USA, May 1999, pp. 173-182

36. MacBeth M, McGuigan K, Hatcher P. Executing Java threads in parallel in a distributed-memory environment. In *In Proceedings of the CASCON'98; Missisauga, ON, 1998, pp. 40-54. Published by IBM*

Canada and the National Research Council of Canada.. November, 1998.

37. Martin P, Silva L, Silva J. 1998. A Java Interface to MPI. In *Proceeding of the* $5^{th\cdot}$ *European PVM/MPI Users Group Meeting*; Liverpool, UK, September 1998.
38. McCarthy J. 1960. Recursive functions of symbolic expressions and their computation by machine. *Communications of the ACM* 1960; 3:184-195.
39. Message Passing Interface Forum. *http://www.mpi-forum.org.* Accessed on December, 25.
40. 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, California, USA, June 1999.
41. 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.
42. Philippsen M, Zenger M. 1997. JavaParty - Transparent Remote Objects in Java. In *Concurrency: Practice and Experience* 1997; 9(11): 1125-1242.
43. Philippsen M, Haumacher M, Nester C. 2000. More Efficient Serialization and RMI for Java. *Concurrency: Practice and Experience* 2000; 12(7):495-518.
44. Rangarajan M, Iftode L. 2000. Software Distributed Shared Memory over Virtual Interface Architecture: Implementation and Performance. In *Proceedings of the 4th Annual Linux Showcase and Conference, Atlanta, October 10-14, 2000, Atlanta, Georgia, USA.*
45. Thurman D. jPVM: A native methods interface to PVM for the Java^{TM} platform. *http://www.chmsr.gatech.edu/jPVM/.* Accessed on December, 25.
46. Transaction Processing Performance Council. *http://www.tpc.org.* Accessed on December, 25.
47. Tyma P. 1998. Why are we using Java again? *Communications of the ACM* 1998;41(6): 38-41.
48. 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.*
49. 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.
50. Compaq Corporation, Intel Corporation, Microsoft Corporation. 1997. Virtual Interface Architecture Specification. Version 1.0. *http://www.viarch.org.* Accessed on December, 25.

51. Wen Y, Carpenter B, Fox G, Zhang G. 1998. Java Data Parallel Extensions with Runtime System Support. In *Proceedings of the ACM 1998 Workshop on Java for High Performance Network Computing*; Palo Alto, California, USA, February 1998.

52. 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.

53. Zhou Y, Iftode L, Li K. 1996. Performance evaluation of two homebased lazy release consistency protocols for shared virtual memory systems. In *Proceedings of the 2nd Symposium on Operating Systems Design and Implementation*, October 1996.

