JPVM: Network Parallel Computing in Java

Adam J. Ferrari
ferrari@cs.virginia.edu

Technical Report CS-97-29
Department of Computer Science
University of Virginia, Charlottesville, VA 22903, USA

December 8, 1997

Abstract

The JPVM library is a software system for explicit message-passing based distributed memory MIMD parallel programming in Java. The library supports an interface similar to the C and Fortran interface provided by the Parallel Virtual Machine (PVM) system, but with syntax and semantics modifications afforded by Java and better matched to Java programming styles. The similarity between JPVM and the widely used PVM system supports a quick learning curve for experienced PVM programmers, thus making the JPVM system an accessible, low-investment target for migrating parallel applications to the Java platform. At the same time, JPVM offers novel features not found in standard PVM such as thread safety, multiple communication end-points per task, and default-case direct message routing. JPVM is implemented entirely in Java, and is thus highly portable among platforms supporting some version of the Java Virtual Machine. This feature opens up the possibility of utilizing resources commonly excluded from network parallel computing systems such as Macintosh and Windows-NT based systems. Initial applications performance results achieved with a prototype JPVM system indicate that the Java-implemented approach can offer good performance at appropriately coarse granularities.

1 Introduction

The use of heterogeneous collections of computing systems interconnected by one or more networks as a single logical computational resource has become a wide-spread approach to high-performance computing. Network parallel computing systems allow individual applications to harness the aggregate power of the increasingly powerful, well-networked, heterogeneous, and often largely under-utilized collections of resources available to many users [1]. In this paper, we describe a network parallel computing software system for use with and implemented in the Java language. Because of its mandated platform independence and uniform interface to system services, Java provides an attractive environment for the implementation of both network parallel applications and the system software needed to support them.

Although numerous software systems support some form of network parallel computing, the majority of use has thus far been based on a small set of popular packages that provide an explicit message-passing, distributed memory MIMD programming model such as Parallel Virtual Machine (PVM) [4], and the Message Passing Interface (MPI) [6]. These software systems support simple, portable library interfaces for typical high-performance computing languages such as C and Fortran. For example, PVM provides the programmer with a library routines to perform task creation, data marshalling, and asynchronous message passing. In addition, PVM provides tools for specifying and managing a collection of hosts on which applications will execute.

Results obtained with network parallel computing systems have been encouraging. For example, a performance study of the NAS benchmark suite implemented in PVM demonstrated that relatively small clusters of workstations could provide performance comparable to significantly more expensive supercomputers [9]. However, the utilization of distributed, heterogeneous, shared resources

connected by commodity networks as a single, virtual parallel computer poses serious problems for both the application and system software programmer. For example, from the application perspective, it has been found that successful network parallel programs will almost always exhibit medium to coarse granularity, and will be tolerant of network latency, for example through the use of a send-ahead programming style. These attributes can be difficult to achieve within some applications. From the system programmer perspective, heterogeneity results in difficult problems such as task to platform matching and system portability.

The Java language provides a number of features that appear to be promising tools for addressing some of the inherent problems associated with network parallel programming. For example, from the application perspective, Java provides a portable, uniform interface to threads. Using threads instead of traditional heavyweight processes has been found to be an avenue for increasing latency tolerance and allowing finer-grained computations to achieve good performance in distributed memory parallel processing environments [3]. From the system implementation perspective, Java supports a high degree of code portability and a uniform API for operating system services such as network communications.

The JPVM (Java Parallel Virtual Machine) library is a software system for explicit message-passing based distributed memory MIMD parallel programming in Java. The library supports an interface similar to the C and Fortran interfaces provided by the Parallel Virtual Machine (PVM) system, but with syntax and semantics enhancements afforded by Java and better matched to Java programming styles. The similarity between JPVM and the widely used PVM system supports a quick learning curve for experienced PVM programmers, thus making the JPVM system an accessible, low-investment target for migrating parallel applications to the Java platform. At the same time, JPVM offers novel features not found in standard PVM such as thread safety, multiple communication end-points per task, and default-case direct message routing. JPVM is implemented entirely in Java, and is thus highly portable among platforms supporting some version of the Java Virtual Machine. This feature opens up the possibility of utilizing resources commonly excluded from network parallel computing systems such as Macintosh and Windows-NT based systems.

In this paper we describe the JPVM system interface, implementation, and performance. In Section 2 we describe the interface to the JPVM system including the programming model and interactive console. In Section 3 we describe key features of a complete, working JPVM prototype implementation. In Section 4 we describe initial performance results achieved with JPVM. In Section 5 we describe related systems, and in Section 6 we conclude.

2 Interface

The programming interface provided by the JPVM system is intentionally similar to that supported by the PVM system, with the addition of enhancements to better exploit the potential benefits of Java as a language for implementing network parallel applications. As in PVM, the programmer decomposes the problem to be solved into a set of cooperating sequential task implementations. These sequential tasks execute on a collection of available processors and invoke special library routines to control the creation of additional tasks and to pass messages among tasks. In JPVM, task implementations are coded in Java, and support for task creation and message passing is provided by the JPVM library.

The central interface through which most JPVM interaction takes place is exported by the **`jpvmEnvironment`** Java class. Instances of this class are declared by JPVM tasks to connect to and interact with the JPVM system and other tasks executing within the system. Objects of this class represent communications end-points within the system, and are identified by system-wide unique identifiers of the opaque type **`jpvmTaskId`** (analogous to a PVM task identifier). Whereas standard

PVM restricts each task to having a single communications end-point (and correspondingly a single task identifier), JPVM allows tasks to maintain a logically unlimited number of communication connections simply by allocating multiple instances of **jpvmEnvironment**. The ability to contain multiple communication end-points in a single task simplifies the process of developing separate linkable modules that need to perform communication. First-class separation of communication streams eliminates the need to artificially distinguish between messages intended for different modules.

After an instance of **jpvmEnvironment** is allocated, its containing task can invoke basic JPVM services such as task creation and message passing. For example, a task can determine its identity for JPVM communication by invoking the **pvm_mytid()** method. A task can detach a **jpvmEnvironment** from the JPVM system by invoking the **pvm_exit()** method. The basic **jpvmEnvironment** interface is depicted in Figure 1.

```
class jpvmEnvironment {
   public jpvmEnvironment(); // Constructor registers task with the JPVM system
   public void pvm_exit();

   // Identity:
   public jpvmTaskId pvm_mytid();
   public jpvmTaskId pvm_parent();

   // Task creation:
   public int pvm_spawn(String task_name, int num, jpvmTaskId tids[]);

   // Send messages:
   public void pvm_send(jpvmBuffer buf, jpvmTaskId tid, int tag);
   public void pvm_mcast(jpvmBuffer buf, jpvmTaskId tids[], int ntids, int tag);

   // Receive messages, blocking (non-blocking versions not depicted):
   public jpvmMessage pvm_recv(jpvmTaskId tid, int tag);
   public jpvmMessage pvm_recv(jpvmTaskId tid);
   public jpvmMessage pvm_recv(int tag);
   public jpvmMessage pvm_recv();

   //Probe for message availability:
   public boolean pvm_probe(jpvmTaskId tid, int tag);
   public boolean pvm_probe(jpvmTaskId tid);
   public boolean pvm_probe(int tag);
   public boolean pvm_probe();

   // System configuration and control:
   public jpvmConfiguration pvm_config();
   public jpvmTaskStatus pvm_tasks(jpvmConfiguration conf, int which);
   public void pvm_halt();
};
```

Figure 1. **jpvmEnvironment** interface excerpt.

2.1 Task Creation

The first action performed by a typical JPVM program is the creation of additional tasks to achieve parallel execution. Task creation in JPVM is supported by the **pvm_spawn()** method, which takes a string parameter indicating the name of a valid Java class visible in the **CLASSPATH** environment variable, as well as the number of tasks to spawn and an array into which the

jpvmTaskIds of the newly created tasks will be placed on successful return. Each task created through **pvm_spawn()** executes in its own instance of the Java Virtual Machine, avoiding issues such as conflicting usage of system services among tasks. These newly created instances of the Java Virtual Machine are placed throughout the set of processors available to JPVM, and each runs an object of the specified Java class.

The identity of newly spawned tasks gives rise to a basic problem for JPVM. In PVM, the identity of a task is clear since tasks have a single communication end-point and thus a single task identifier. In JPVM, tasks can have any number of communication end-points, and thus it is unclear what identity for a newly spawned task should be returned to the parent (i.e. the spawning task). One simple solution would be to remove the return of task identifiers from the spawn interface. Spawned tasks could communicate their identities to their parents explicitly to enable communication. In order to retain an interface similar to PVM, and to avoid additional application code in the common case of a single identity per task, JPVM instead addresses this issue by returning the identity of the first **jpvmEnvironment** allocated in each newly spawned task. For standard, single-identity tasks, this provides the familiar PVM style of identifying newly spawned tasks.

2.2 Message Passing

Message passing in JPVM is performed using the **pvm_send()** and **pvm_recv()** methods of the **jpvmEnvironment** class. However, before data can be sent, it must be collected into a **jpvmBuffer** object. Analogous to PVM buffers, **jpvmBuffer** objects are the message content containers of JPVM. The **jpvmBuffer** interface (depicted in Figure 2) contains two basic groups of methods: those to pack data into a buffer, and those to extract data from a buffer. Where possible, overloading is used in the interface to simplify application code. Scalar and vector pack and unpack operations are provided for all basic Java types as well as **String** and **jpvmTaskId** objects.

```
class jpvmBuffer {
    public jpvmBuffer();

    public void pack(int v[], int n, int stride);
    public void pack(int s);
    public void pack(float v[], int n, int stride);
    public void pack(float s);
                    . . .
    public void unpack(int v[], int n, int stride);
    public int upkint();
    public void unpack(float v[], int n, int stride);
    public float upkfloat();
                    . . .
};
```

Figure 2. **jpvmBuffer** interface excerpt.

An important difference between JPVM buffers and standard PVM buffers is the explicit nature of the JPVM buffer data structure. In standard PVM, send and receive buffers are manipulated implicitly by the **pvm_pk*()** and **pvm_upk*()** library routines. Whereas routines are provided to manage multiple buffers, the implicit nature of the buffer being packed or unpacked gives rise to a difficult interaction with threaded programs. In PVM, if a thread is packing one buffer, and a second thread is packing another buffer, the threads must explicitly set the send buffer before each pack routine (since the other thread may have run and set the send buffer since the current thread's last pack). Furthermore, a synchronization mechanism such as a lock must be used to ensure that the send buffer is not reset by another thread between being set and packed by the current thread. This is a complex and unnecessary requirement to place on applications programs. In JPVM this problem is addressed

simply and directly through the use of explicit buffers. Pack and unpack operations manipulate the buffer object on which they are invoked, leading to a simplified interface for dealing with threaded tasks. This is one example of JPVM's changes to the standard PVM interface to better support common Java programming styles such as the use of threads.

After the contents of a message have been marshalled into a **jpvmBuffer** object, the buffer can be sent to any task in the JPVM system using the **pvm_send()** method of the **jpvmEnvironment** class. Besides taking the buffer to be sent, **pvm_send()** also requires the identity of the task to which to the message should be delivered (in the form of a **jpvmTaskId**) and an integer identification number for the message called the message tag. The send operation is asynchronous—the sending task proceeds immediately after the send is initiated, regardless of when (or if) the message is received.

To receive messages, tasks must execute the **pvm_recv()** method of the **jpvmEnvironment** class. Using the various versions of **pvm_recv()** depicted in Figure 1, tasks can request messages based on the identity of the message sender, the identification number (tag) of the message, both of these, or neither (i.e. receive any message). Receive operations block until a message of the requested type is available (non-blocking **pvm_nrecv()** versions are also supported, but are not depicted in Figure 1), at which point a **jpvmMessage** structure is returned containing the received message. Besides containing a **jpvmBuffer** with the message contents, a received **jpvmMessage** object contains an indication of the identity of the sender of the message as well as the identification number of the message, as depicted in Figure 3.

```
class jpvmMessage {
    public int          messageTag;
    public jpvmTaskId   sourceTid;
    public jpvmBuffer   buffer;
};
```

Figure 3. **jpvmMessage** data structure.

2.3 Program Example

A brief example of a JPVM program is depicted in Figure 4. This task represents the master in a simple master/slave program organization. Note that the task is implemented as a standard Java application object with a public **main** method. The program first creates a connection to the JPVM system on line 5 where it allocates a **jpvmEnvironment** object. It then spawns a set of worker tasks on line 9. Note, for this spawn to succeed, a valid Java class must be stored in the file "**worker.class**" located in a directory listed in the **CLASSPATH** variable. The spawn operation creates **N** new Java Virtual Machine processes, each executing an instance of the class **worker**, and returns the identity of these tasks' first **jpvmEnvironment** connections.

The loop on lines 11-16 demonstrates the creation of a **jpvmBuffer**, message marshalling, and use of the **pvm_send()** method to send a message to each newly spawned task. This loop might correspond to farming out work to a group of worker tasks. Correspondingly, the loop on lines 17-21 demonstrates the receipt of messages from each of the worker tasks, which might correspond to gathering results. On line 22 the program invokes the **pvm_exit()** method to disconnect from the JPVM system.

```
1  import jpvm.*;
2  class example {
3     public static void main(String args[]) {
4        try {
5           jpvmEnvironment jpvm = new jpvmEnvironment();
6
7           // Spawn N worker tasks
8           jpvmTaskId tids[] = new jpvmTaskId[N];
9           jpvm.pvm_spawn("worker",N,tids);
10
11          for (int i=0;i<N; i++) { // Farm out work
12             jpvmBuffer buf = new jpvmBuffer();
13             int work = getWork(i);
14             buf.pack(data);
15             jpvm.pvm_send(buf, tids[i], 123);
16          }
17          for (int i=0;i<N; i++) { // Receive results
18             jpvmMessage message = jpvm.pvm_recv(tids[i]);
19             int result = message.buffer.upkint();
20             processResult(result);
21          }
22          jpvm.pvm_exit();
23       }
24       catch (jpvmException jpe) {
25          System.out.println("Error - jpvm exception");
26       }
27    }
28 };
```

Figure 4. JPVM programming example.

2.4 System Configuration

The above discussion of the programming model describes the basic syntax and semantics of the JPVM library. However, the JPVM library routines require run-time support during execution in the form of a set of JPVM daemon processes running on the available collection of processors. These daemon processes are required primarily to support task creation (unlike in standard PVM, JPVM daemon processes do not perform message routing). JPVM system configuration is a two-phase process. First, daemon processes must be started manually on all hosts of interest. The daemon program is provided in the form of a java class (**`jpvmDaemon`**), so daemon startup simply involves running Java Virtual Machine processes to execute the daemon class instances. After the daemons are started, they must be notified of one another's existence. This is accomplished through an interactive JPVM console program. Again, the console program is a java class (**`jpvmConsole`**) and should be started in a Java Virtual Machine process.

Besides adding hosts, the JPVM console can be used to list the hosts available to the system and JPVM tasks running in the system. An example console session is depicted in Figure 5.

3 Implementation

As was mentioned previously, JPVM is implemented entirely in Java. Although current execution environments for Java limit performance significantly, the use of Java as a system implementation language provides a number of distinct advantages over commonly used system implementation languages such as C. For example, the use of threads in a communications system is an attractive approach for dealing with the asynchronous activities involved in this type of software. However,

```
$ java jpvm.jpvmConsole
jpvm> conf
4 hosts:
        stonesoup00.cs.virginia.edu
        stonesoup01.cs.virginia.edu
        stonesoup02.cs.virginia.edu
        stonesoup03.cs.virginia.edu
jpvm> ps
stonesoup00.cs.virginia.edu, 2 tasks:
        (command line jpvm task)
        jpvm console
stonesoup01.cs.virginia.edu, 1 tasks:
        mat_mult
stonesoup02.cs.virginia.edu, 1 tasks:
        mat_mult
stonesoup03.cs.virginia.edu, 1 tasks:
        mat_mult
jpvm>
```

Figure 5. Example interactive JPVM console session.

porting threads-based systems implemented in traditional languages such as C can be a complex and error-prone task. Java reduces the risks associated with employing threads in system designs by providing a uniform, portable threads interface and implementation across all supported platforms. Another example of the portability advantage of Java is network APIs. Although network-based systems such as standard PVM are generally portable among Unix variants, they are more difficult to port to other readily available environments such as PCs running Windows NT. For example, although a wide variety of PVM ports for Unix-based systems have been available for close to a decade, only in the past year has a Windows based implementation become available. Java essentially eliminates this issue by providing a uniform, portable interface to operating system services such as network communication.

Given the portability advantage of Java, we chose to implement the JPVM interface entirely in Java. In this section we discuss some of the core features of the JPVM system implementation.

3.1 Communications Implementation

In standard PVM, the use of direct task-to-task TCP connections has been found to significantly outperform the older UDP-based daemon-routed message passing implementation [9]. Thus, the current internal JPVM message passing implementation is based on direct task-to-task communication over TCP sockets using the Java object serialization interface for message transfer. Each **`jpvmEnvironment`** instance creates a server socket during initialization and embeds the host name and port number needed to connect to it within its **`jpvmTaskId`** identity. Thus, when JPVM task X wishes to send a message to JPVM task Y, it simply connects to Y using the host and port contained within Y's task identifier, and sends a message over the newly created TCP connection.

Internally, JPVM uses threads to manage connections and message delivery. Each **`jpvmEnvironment`** creates a dedicated thread to listen for and accept peer connections, as depicted in Figure 6. When a connection is accepted by this special dedicated connection management thread, it creates a new message receiver thread dedicated to the new connection. This receiver thread blocks for messages on its connection. As it receives messages, it enqueues them on an internal message queue, as illustrated in Figure 6. Thus when user threads perform **`pvm_recv()`** operations, instead of performing any network input directly, they simply consult the internal message queue for messages of the appropriate type.

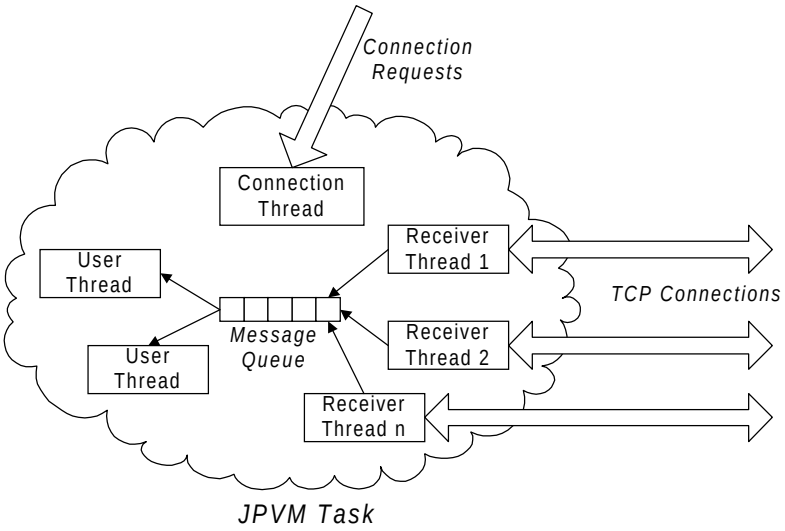

Figure 6. JPVM communication implementation.

An important attribute of the JPVM communications system implementation is thread safety. As depicted in Figure 6, multiple user threads can concurrently perform **`pvm_recv()`** operations—the internal message queue structure is a synchronized data type that may safely be manipulated by any number of reader (user) and writer (internal receiver) threads. Similarly, **`pvm_send()`** operations are synchronized to support thread safety. Threads can be employed in a variety of design schemes in JPVM programs. For example, a traditional threaded server JPVM task is one possibility. Another possibility is the use of threads as the basic units of parallel execution, as in TPVM [3]. In this scheme, each thread would be implemented as a normal JPVM task with its own **`jpvmEnvironment`** instance.

3.2 Daemon Services Implementation

The ability to spawn new tasks, to determine and extend system configuration, and to determine the set of tasks executing in the system is based on services provided by JPVM daemon processes executing on each host in the JPVM system. The current implementation of the JPVM daemon processes is layered on the standard JPVM communication mechanism—i.e. the JPVM daemon implementation is simply a normal JPVM program employing the interface described in Section 2. The daemon program employs a server style organization, executing a loop that repeatedly blocks for and then services a new request. For most client requests (e.g. task status, system configuration information) the daemon can respond immediately. However, for task creation requests the daemon creates a thread to start the new local process. This design prevents slow response to other client requests while tasks are being created.

4 Performance

To evaluate the performance of the current JPVM implementation we conducted a set of experiments to quantify the costs of the basic library primitives and to measure actual application performance. The testbed used for these experiments was a set of five dual-processor Pentium Pro hosts connected by 10 megabit ethernet. Each host in the system contained two 200 MHz Pentium Pro processors and 128 MB of memory, and was running Linux 2.0. The version of Java used was the Linux JDK version 1.1.3.

Our first set of experiments measured the cost of task creation. In Table 1 we present the time to create various numbers of tasks on a single host system and in the full five host system. Since each host is a dual processor machine, the difference in time to create one or two hosts is negligible. The cost for task creation is not small—when creating more than one task, approximately 0.5 seconds per task in the single host system, and between 0.5 and 1.5 seconds per task in the multi-host system are required. Spawn is somewhat less expensive for small numbers of tasks in the single host system because less communication is required. For larger numbers of tasks the multi-host system begins to perform better as it can amortize communications costs by parallelizing the creation of a set of tasks.

tasks	1 host	5 hosts
1	886	2437
2	1075	2509
4	2092	2574
8	4042	3683
16	7938	6156

Table 1. Task creation times, milliseconds

size	int	byte
1 byte	240	224
1 KB	271	243
10 KB	293	282
100 KB	1160	846
1 MB	9309	7535

Table 2. Round-trip message costs, milliseconds

Our next set of measurements examined the communication costs associated with JPVM. In Table 2 we present round-trip message times between two tasks for various message sizes. We performed these measurements on messages containing integers and raw bytes to examine the overhead associated with the requirement of masking byte ordering differences. As might be expected, the message passing overheads introduced by the Java implemented JPVM system are very high—an order of magnitude of latency is added, and only about an eighth of the available bandwidth is utilized.

Given the high costs associated with both of the basic JPVM primitives, it is tempting to conclude that the overhead introduced by the strategy of using a Java-implemented system as runtime support for parallel Java applications outweighs the potential benefits. In fact, the high costs of these operations does essentially rule out JPVM as a platform for network-intensive applications. However, for applications that exhibit medium to coarse granularities, and also exhibit some tolerance of network latencies, JPVM may provide sufficiently efficient services to allow good speedup.

To verify this claim, we measured the performance of a parallel matrix multiplication algorithm implemented in JPVM and compared to a sequential version of the program also implemented in Java. In Table 3 we present the results of these experiments for three problem sizes. Times are presented for both the complete program (including task creation time), and for the multiplication algorithm alone (excluding task creation time). As might be expected, at the smallest problem size, poor speedup is achieved for the multiplication algorithm, and slowdown is observed for the program as a whole. This problem size is simply too fine grained given the basic costs associated with JPVM. At the middle problem size we find that the multiplication algorithm begins to speed up well, but the program as a whole has poor efficiency due to the high cost of task creation. Finally, at the largest problem size, both the multiplication algorithm and the program as a whole speed up well. As expected, better speedups are achieved for the program when task creation costs are excluded. Although this may seem like an unfair comparison, it provides valuable insight that for long-running JPVM programs that can amortize task creation costs, good efficiency is possible for the compute/communicate phases of the program.

problem size	tasks	multiply time	speedup	total time	speedup
128	*1*	3.2	-	3.2	-
	4	2.2	1.4	6.6	0.5
	9	1.5	2.1	6.7	0.5
256	*1*	26.7	-	26.7	-
	4	7.7	3.5	10.3	2.6
	9	4.6	5.9	8.9	3.0
512	*1*	223.3	-	223.3	-
	4	59.7	3.7	64.3	3.5
	9	28.9	7.7	33.5	6.7

Table 3. Matrix multiply performance, times in seconds

5 Related Work

A number of other systems have been developed to support network parallel programming in Java. One common approach to this problem is the use of volunteer-based systems such as Bayanihan [7] and Javelin [2]. As opposed to JPVM which is based on stand-alone Java applications, these systems are based on Java applets that execute within the context of a web browser. In these systems, applications are decomposed into sub-tasks that can be downloaded in the form of Java applets by clients who wish to volunteer computing resources. When the task completes, its result is uploaded to the server. These approaches have a number of attractive properties, foremost among which is the possibility of employing the vast array of processing power available in the form of client machines connected to the web. The primary drawback of these approaches is the significant restrictions placed on communications by the Java security model for applets. The restriction that downloaded applets may communicate only with their server essentially limits these systems to applications that can be decomposed into non-communicating, coarse-grained, completely independent functional tasks. Although this is possible for many applications such as parameter space studies, it rules out most potentially successful network parallel applications.

A system similar to JPVM in its programming interface and model is the JavaPVM library [8]. This system also provides a PVM-like interface for Java applications. The primary difference between JavaPVM and JPVM is in implementation. Unlike JPVM which is implemented entirely in Java, JavaPVM is based on the Java Native Methods mechanism, providing native method wrappers around the existing standard PVM routines. This approach has a number of advantages. First, this approach takes clear advantage of software reuse—the effort to produce the PVM library is not duplicated, and advances in the standard PVM system can be tracked closely. Also, since JavaPVM is based on native methods wrappers to the standard PVM library, JavaPVM programs are interoperable with standard PVM programs written in C and Fortran. Furthermore, by using native methods JavaPVM offers a clear performance advantage in the short term. Currently, systems implemented in lower-level languages such as C can easily outperform Java-based implementation. However, the JavaPVM approach is not without drawbacks. First, by using native methods, this system is limited in its portability. JavaPVM can be used only on platforms where standard PVM is available—for example, JavaPVM isn't currently supported on Windows-NT or Macintosh systems. Furthermore, in being a simple wrapper around the standard PVM implementation, JavaPVM is limited by the semantic and syntactic features of PVM. For example, JavaPVM programs retain the single communication end-point PVM model, the thread-unfriendly PVM buffer interface, and so on.

Another software package for network parallel computing in Java is the IceT system [5]. This

system addresses a number of concerns not covered by the JPVM design. Whereas JPVM allows a single user to combine resources on which that user has access privileges, IceT is an inherently collaborative, multi-user environment. IceT provides mechanisms by which resources can be made available to the system for use by users who do not have log-on privileges. The attractive goals of this metacomputing-based approach include better utilization of larger resources based, as well as an enhanced collaborative environment for high performance computing. Among the challenges introduced by this idea are the numerous security issues introduced by code-upload capabilities for non-privileged uses.

6 Conclusions

We have described the JPVM system, both in terms of programming model and interface, interactive interface, and implementation. This system combines the goals of supporting network parallel programming in Java, providing a familiar and proven-effective programming interface, and taking advantage of the attractive features of Java as a system implementation language. The combination of Java as an applications programming language and system implementation languages allows the support of a number of powerful features in JPVM, including thread safety and multiple communication end-points per process. Initial performance experiments with the system indicate that the costs associated with the basic JPVM primitives are quite high, but his is not unexpected given the use of Java for system implementation. However, we have demonstrated that the system can support good speedup for applications that exhibit an appropriately coarse level of granularity, tolerance of network latency, and ability to amortize task creation costs. Furthermore, these are issues already well-known applications programers who employ network parallel programming. Given this observation, we argue that the added costs associated with a Java-based system do not outweigh the benefits of the added features, portability, and programming flexibility afforded by Java.

References

[1] T.E. Anderson, D.E. Culler, D.A. Patterson, and the NOW team, "A Case for NOW (networks of Workstations)," *IEEE Micro*, vol. 15, no. 1, pp. 54-64, February, 1995.

[2] P. Cappello, B.O. Christiansen, M.F. Ionescu, M.O. Neary, K.E. Schauser, and D. Wu, "Javelin: Internet-based Parallel Computing Using Java," *ACM Workshop on Java for Science and Engineering Computation*, June, 1997.

[3] A.J. Ferrari and V.S. Sunderam, "Multiparadigm Distributed Computing with TPVM," *Journal of Concurrency, Practice and Experience*, (to appear).

[4] A. Geist, A Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V.S. Sunderam, PVM: Parallel Virtual Machine, MIT Press, 1994.

[5] P.A. Gray and V.S. Sunderam, "IceT: Distributed Computing and Java," available from: **`http://www.mathcs.emory.edu/~gray/abstract7.html`**

[6] W. Gropp, E. Lusk, and A. Skjellum, Using MPI: Portable Parallel Programming with the Message-Passing Interface, MIT Press, 1994.

[7] L.F.G. Sarmenta, "Bayanihan: Web-Based Volunteer Computing Using Java," available from: **`http://www.cag.lcs.mit.edu/bayanihan/`**

[8] D. Thurman, *JavaPVM*, available from: **`http://www.isye.gatech.edu/chmsr/JavaPVM/`**

[9] S. White, A. Ålund, and V.S. Sunderam, "Performance of the NAS Parallel Benchmarks on PVM Based Networks," *Journal of Parallel and Distributed Computing*, vol. 26, no. 1, pp. 61-71, April 1995.