JPVM: Network Parallel Computing in Java

Adam Ferrari
Department of Computer Science
University of Virginia, Charlottesville, VA 22903, USA
ferrari@virginia.edu

Abstract

The JPVM library is a software system for explicit message-passing based parallel programming in Java. The library supports an interface similar to the C and Fortran interface of the Parallel Virtual Machine (PVM) system, with modifications to better suit Java programming styles. The similarity between JPVM and the widely used PVM system supports a quick learning curve for experienced PVM programmers. 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 a version of the Java Virtual Machine. This feature opens up the possibility of utilizing resources such as Macintosh and Windows based workstations, which are commonly excluded from network parallel computing 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 networks of heterogeneous computing systems interconnected as a single logical computational resource has become a widespread approach to high-performance computing. Network parallel computing systems allow individual applications to harness the aggregate power of increasingly powerful, well-networked, heterogeneous, and often largely under-utilized collections of resources available to many users [1]. Results obtained with network parallel computing systems have been encouraging [10], but 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, but these attributes can be difficult to achieve in some applications. From the system implementation 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 heavy-weight 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 [5]. 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 [4] 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 [6], 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 such as Macintosh and Windows based workstations, which are commonly excluded from network parallel computing systems.

2 Programming Interface

The programming interface provided by the JPVM system is intentionally similar to that supported by PVM, with the addition of enhancements to better exploit the potential benefits of Java as a language for network parallel programming. 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 (as applications with a static public `main()` method), and support for task creation and message passing is provided by a library of JPVM Java classes.

The most basic JPVM interface is exported by the `jpvmEnvironment` class (see Figure 1), instances of which are declared by JPVM tasks to connect to and interact with the JPVM system and other tasks. 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). JPVM allows tasks to maintain a logically unlimited number of communication connections simply by allocating multiple instances of `jpvmEnvironment`. This feature simplifies the process of developing separate linkable modules that need to perform communication.

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. 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.

Message passing in JPVM is performed using the `pvm_send()` and `pvm_recv()` methods of the `jpvmEnvironment` class. 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 (see 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. The JPVM buffer

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

    public void pvm_exit();
    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();
};
```

Figure 1: Excerpt from the `jpvmEnvironment` interface.

interface differs from standard PVM in its use of explicit buffer objects (as compared to the use of implicit send and receive buffers in PVM). This design is an example of JPVM's changes to the standard PVM interface to better support multithreaded programming.

```
class jpvmBuffer {
    public jpvmBuffer();
    public void  pack(float v[], int n, int stride);
    public void  pack(float s);
    public void  unpack(float v[], int n, int stride);
    public float upkfloat();

    // ...Analogous functions for other data types
};
```

Figure 2: Excerpt from the `jpvmBuffer` interface.

After the contents of a message have been marshaled into a `jpvmBuffer` object, the buffer can be sent to any task in the JPVM system using the asynchronous `pvm_send()` method of the `jpvmEnvironment` class. Parameters to `pvm_send()` specify the buffer to be sent, the identity of the task to which to the message should be delivered, and an integer identification number for the message (called the message tag).

To receive messages, tasks call the `pvm_recv()` method of the `jpvmEnvironment` class. Using the various versions of `pvm_recv()`, 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, at which point a `jpvmMessage` structure is returned containing the received message (non-blocking versions are also supported, but are not depicted in Figure 1). Besides containing a `jpvmBuffer` with the message contents, a received `jpvmMessage` object contains the identity of the sender of the message as well as the message tag.

3 Implementation

Although current execution environments for Java limit performance significantly, the use of Java as a system implementation language provides a number of distinct advantages, foremost amongst which is the ability to employ services such as threads and sockets through a portable interface across a wide range of platforms. In this section we describe the key features of our entirely Java-based JPVM implementation.

The JPVM message passing implementation is based on communication over TCP sockets. The use of direct task-to-task TCP connections has been found to significantly outperform the older UDP-based daemon-routed message passing implementation in PVM [10]. 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. 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 3. When a connection is accepted by this special dedicated connection management thread, it creates a new message receiver thread dedicated to the new connection. As the receiver thread receives messages, it enqueues them on an internal message queue (illustrated in Figure 3). Thus, when user threads perform `pvm_recv()` operations, instead of performing any network input directly, they simply consult the internal message queue. An important attribute of the JPVM communications system implementation is thread safety. As depicted in Figure 3, 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. The `pvm_send()` operations are also synchronized to support thread safety.

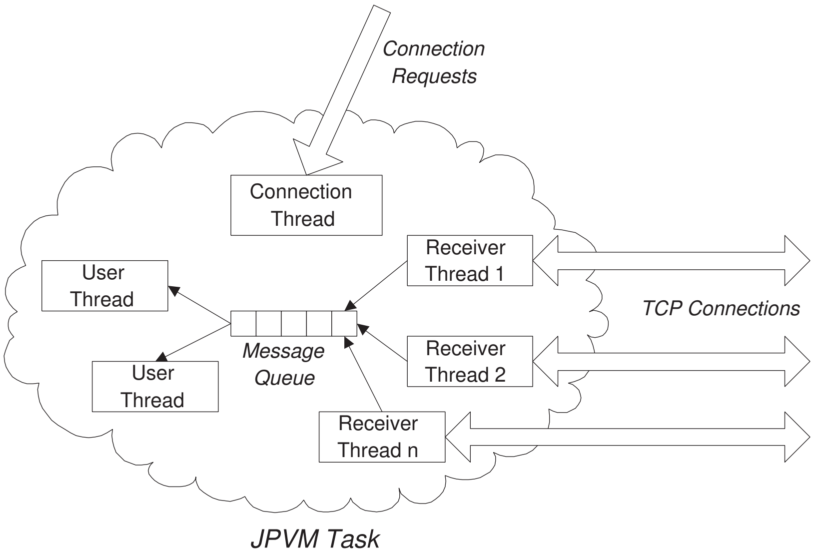

Figure 3: JPVM message-passing implementation.

The ability to spawn new tasks, to determine and extend system configuration, and to determine which tasks are executing in the system is based on services provided by a set of JPVM daemon processes executing on each host in the system. The current daemon implementation is layered on the standard JPVM communication mechanism–i.e. the implementation simply comprises a normal JPVM program using the interface described in Section 2. The daemon employs a server style organization, repeatedly blocking for and servicing new requests. 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 avoiding slow response to other client requests while tasks are being created. The ability to use multithreading in the design of responsive server processes is one example of the value of JPVM's thread-safe interface.

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 100 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 [2].

In Figure 4 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. In all cases, the high cost of task creation is primarily due to the time required to start and initialize a new instance of the JVM to host the task object.

Tasks	1 Host	5 Hosts
1	886	2437
2	1075	2509
4	2092	2574
8	4042	3683
16	7938	6156

Figure 4: Task creation time, milliseconds.

In Figure 5 we present round-trip message times between two tasks for a range of 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 somewhat high–approximately two orders of magnitude of latency are added to small messages as compared to the round-trip time reported by the Unix `ping` utility. Bandwidth is not as badly affected; for large messages, we achieve approximately half of the bandwidth measured for native-code processes using the raw Unix TCP sockets interface for communication.

	Integer	Byte
4 B	32.3	30.3
1 KB	55.8	52.9
10 KB	68.0	62.4
100 KB	325.5	309.4
1 MB	2940.2	2386.9

Figure 5: Round-trip message-passing time, milliseconds.

Given the relatively high costs associated with 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, or programs with a high dynamic task-creation rate. 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 Figure 6 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,

Problem Size	Tasks	Multiply Time	Speedup	Total Time	Speedup
128	1	3.2	N/A	3.2	N/A
128	4	2.2	1.4	6.6	0.5
128	9	1.1	2.1	6.7	0.5
256	1	26.7	N/A	26.7	N/A
256	4	7.7	3.5	10.3	2.6
256	9	4.6	5.9	8.9	3.0
512	1	223.3	N/A	223.3	N/A
512	4	59.7	3.7	64.3	3.5
512	9	28.9	7.7	33.5	6.7

Figure 6: Matrix multiply performance, times in seconds.

better speedups are achieved for the program when task creation costs are excluded. Although this may seem like an unfair comparison, it demonstrates that for long-running JPVM programs that can amortize task creation costs, good efficiency is possible for the compute/communicate phases of the program.

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 [8] and Javelin [3]. As opposed to JPVM which is based on stand-alone Java applications, these systems are based on the decomposition of programs into applets that are downloaded by volunteer clients, execute within the context of a web browser, and solve some sub-portion of a larger problem. These approaches are attractive in that they can potentially employ 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 restriction placed on communications by the Java security model for applets, which rules out many potentially successful network parallel applications.

A system similar to JPVM in its programming interface and model is the JavaPVM library [9]. 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. Whereas this approach has advantages in terms of performance (at least in the short term) and interoperability, by using native methods, this system is restricted in its portability–JavaPVM can be used only on platforms where standard PVM is available. 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 less thread-friendly buffer interface provided by PVM.

Another software package for network parallel computing in Java is the IceT system [7]. 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 resource bases, 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 this is not unexpected given the use of Java for system implementation. However, we have found 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 to applications programmers 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] S. Byrne, Linux JDK, available from: `http://www.java-linux.org/`

[3] 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.

[4] A.J. Ferrari, JPVM, available from: `http://www.cs.virginia.edu/jpvm`

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

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

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

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

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

[10] S. White, A. Alund, 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.