Communication Performance of Java-based Parallel Virtual Machines

Narendar Yalamanchilli and William Cohen
(yalamanc,cohen)@ece.uah.edu
Department of Electrical and Computer Engineering
University of Alabama in Huntsville,
Huntsville, AL 35899, USA

April 14, 1998

Abstract

Message passing libraries such as Parallel Virtual Machine (PVM) and Message Passing Interface (MPI) provide a common Application Programming Interface (API) to implement parallel programs across multiple computers. Such libraries provide a means to program a collection of normally independent computers to work cooperatively on a single computation. However, for programs written in C and Fortran these collections of machines may provide a heterogenous set of computer architectures, requiring a different executable for each type of architecture.

The Java language offers a potentially machine-independent method of distributing the same code to perform the computations on different computer architectures. The communication performance between processors running Java programs is a crucial issue for this type of application. This paper compares the performance between tradition PVM implemented in C code, Java code interfaced to the traditional PVM libraries (JavaPVM), and Java code that performs functions equivalent to the traditional PVM library (JPVM). The Java implementations are slower, but performance improvements are possible.

1 Introduction

The message passing model, which exchanges data between individual processors with explicit message transmission and reception, has been attractive to parallel application developers because libraries such as the Parallel Virtual Machine (PVM) [1] and the Message Passing Interface (MPI) [2] allow existing clusters of workstations to be used in a cooperative manner to solve a single problem. However, there have been two impediments to writing parallel programs using these libraries: heterogenous processor architectures and interprocessor communication performance.

In many cases the collection of machines used for executing the parallel program are individual workstations. As a result, there may be several different processor architectures represented in the collection, each requiring its own version of the executable program. Although PVM provides a mechanism for archiving and fetching executables for different architectures, it provides little assistance in the initial generation of the executable. The generation of multiple executables leads to problems ensuring that the multiple versions are available to all machines and that each machine obtains the correct version of the code.

The object-oriented programming language Java [3] may provide some relief from this problem. Rather than compiling Java code to a specific processor instruction set, it is compiled into code for the Java Virtual Machine (JVM) [4], an abstract instruction set, and this code is distributed to the processors. The machines that obtain the code in this format can either use an interpreter or can perform another compilation step to generate native code. The Just-In-Time (JIT) compilers that translate the Java Virtual Machine code into native instructions have made significant strides to improve performance, and Java programs have been estimated of being able to obtain 68% of the speed of traditional C code [5]. Thus, Java may provide a viable environment for high-performance programming.

The performance of a parallel program is also influenced by the speed that data can be exchanged between the processors performing the computations. The Java programs communicate between independent machines based on a message passing model. The decision to split computations between machines is based on whether the reduction in time to perform computations on multiple processors is greater than the increased overhead introduced by communication and redundant computations.

In an effort to leverage existing message passing programs, Java classes similar to PVM have been implemented. This paper will examine the performance of two of these PVM-like libraries, JavaPVM [6] and JPVM [7, 8], against the native C implementation of PVM. The gathered data is useful for implementors of communication libraries to determine areas of improvement and is useful for application programmer to determine whether it is profitable to parallelize certain sections of their program. A description of PVM, JavaPVM and JPVM are given in sections 2, 3, and 4, respectively. The test conditions and results are given in section 5. The performances of the message passing libraries are analyzed in section 6. Finally, the results are summarized in section 7.

2 PVM

PVM [1] is a message passing system that connects a heterogeneous collection of computers running the software into one large distributed memory computer. Thus, PVM provides the capability to solve computationally intensive problems by using the aggregate memory and computing power of many computers. A PVM program is implemented as a collection of cooperating tasks (units of parallelism) which access PVM resources through a library of standard interface routines. The library routines create and terminate tasks across heterogeneous clusters of computers as well as provide communication and synchronization between tasks. The PVM libraries, including the ones used for these experiments, are written in C and must be compiled for each machine architecture.

3 JavaPVM

JavaPVM [6] is layered upon the standard distribution of PVM and makes use of Java's capability to call functions written in other languages. The functions written in languages other than Java and called from Java are known as *native* methods. JavaPVM programs use wrappers contained in the class JavaPVM to call the *native* PVM functions, which are written in C. This scheme limits JavaPVM to machines which support both Java and native PVM.

However, there are some advantages to interfacing Java to the original PVM library. Java code could be used to write a portion of the software, e.g. a Graphical User Interface (GUI), while C or Fortran could be used to implement the remainder of the software. PVM would allow communication between the different languages. Thus, JavaPVM would allow programmers to reuse existing code on some processors and not force the programmer to switch all the code to Java.

4 JPVM

JPVM [7, 8] differs from PVM and JavaPVM; it is implemented entirely in Java and uses none of the original PVM code. Unlike JavaPVM, JPVM is not inter-operable with standard PVM. JPVM provides a Java implementation of the PVM daemon (pvmd) and a library for communication. JPVM supports both tasks and threads as basic units of parallelism. Thus, the implementation of JPVM is thread safe. JPVM makes use of the socket interface in the standard API for communication between these parallel units. The Java standards for data formats and built-in parallel programming constructs of Java dramatically simplify the implementation of this communication library.

5 Communication Benchmark

The COMMS1 [9] or ping-pong benchmark was used to measure the basic communication performance of the three PVM software packages. This benchmark sends messages of varying lengths between a master task

Table 1: Round Trip Times for SUN workstation using JDK 1.1.5.

	time in ms				
Bytes Sent	PVM INT	JavaPVM byte	JavaPVM int	JPVM byte	JPVM int
0	.73	4.6	6.8	144	209.
4	.73	3.1	6.3	152	156
12	.74	4.1	5.9	153	158
100	.83	5.1	5.6	131	158
1000	1.6	8.9	8.7	117	188
10000	9.24	21.3	32	112	287
100000	87.2	129	264	278	1740
1000000	872.	1110	2600	1910	11700

and a slave task. On receiving the message, the slave task echos the same message back to the master task.

The amount of time required to send a message from the master task to the slave is assumed to be equal to the time required to send the same message from the slave task to the master task. Thus, the one-way trip time is assumed to be one half of the time required for a message to make a round trip from master task to slave task and back to the master task. By taking a series of measurements for various sizes of messages the communication latency and the communication bandwidth can be calculated.

Versions of the COMMS1 benchmark were written and tested for each of the communication libraries discussed above. The benchmark tests consisted of two tasks, a master task and a slave task, each running on a different computer. When the master task starts running on the local machine, it spawns a slave task on the remote machine. The master task then packs messages with a variable number of data bytes and sends the messages to the slave task. The slave task echos the received messages to the master task. The communication times were measured for messages with payloads of various sizes ranging from no data to 1,000,000 bytes.

Tests were performed on two sets of machines: SUN UltraSPARC 1 workstations and Pentium workstations. Communication tests were performed on two SUN UltraSPARC 1 workstations with 143MHz processors running the Solaris 2.5 operating system connected by a 10 Mb/s Ethernet. The Java code was executed on SUN's 1.1.5 Java Development Kit (JDK). PVM 3.3 compiled with GCC 2.7.2 was used for the PVM and JavaPVM experiments. The same set of communication tests were also performed on the Pentium machines running the Linux 2.0.30 operating system. Each Linux machine had 32MB of memory and an 100MHz Intel Pentium processor. The choice of Java virtual machines on the Pentium workstations were SUN's 1.1.3 Java Development Kit and Kaffe 0.9.1. Runs of the tests under JDK 1.1.5 were attempted on the Pentium workstations, but due to problems with the socket libraries only data for JDK 1.1.3 could be collected. Kaffe 0.9.1 also had some limitation in the library that prevent collection of data for JavaPVM and required the JPVM daemon to be run under JDK 1.1.3. Limitations on the Linux workstations prevented measurement of message sizes greater than 1,000,000 bytes for PVM and JavaPVM.

The round-trip times for the ping-pong benchmark are shown in table 1 and table 2. Table 3 and table 4 show the round-trip latency and time per byte calculated by linear regression for each of the communication libraries in the various runtime environments. Finally, table 5 and table 6 list the effective bandwidth for each of the tests. To determine whether there was a significant difference in overhead for sending different primitive data types, timing data was collected for messages of the same size composed of integers and bytes.

6 Discussion

Table 3 and table 4 summarize the costs associated with the communication packages as latency and time per byte sent. These tables clearly show the performance of each of the communication packages. There are two features immediately apparent in the tables: most of the per byte costs are within a factor of three of the best achieved by PVM and the Java communication packages latencies are one to two orders of magnitude

Table 2: Round Trip Times for Linux.

	time in ms						
	PVM	JavaPVM	JavaPVM	JPVM	JPVM	JPVM	JPVM
Bytes		byte	int	byte	int	byte	int
Sent		JDK	JDK	Kaffe	Kaffe	JDK	JDK
0	1.0	3.1	4.9	21.2	21.0	77.3	79.0
4	1.0	3.2	5.0	20.5	21.4	76.9	77.3
12	1.0	3.5	3.3	22.0	21.9	79.4	79.4
100	1.2	3.5	3.7	21.3	20.9	79.1	80.3
1000	2.5	6.0	6.2	31.1	40.0	83.7	77.9
10000	10.7	21.5	35.0	33.3	45.8	82.9	107.
100000	89.9	138.	295.	194.	196.	256.	533
1000000	—	—	—	2180	2200	1910	4270

Table 3: Cost of sending messages on SUN workstation using JDK 1.1.5.

	time in ms				
	PVM	JavaPVM	JavaPVM	JPVM	JPVM
Bytes Sent		byte	int	byte	int
Latency	.62	7.5	5.9	127	226
Time/byte	.00087	.0011	.0026	0.0018	0.012

Table 4: Cost of sending messages on Linux.

	time in ms						
	PVM	JavaPVM	JavaPVM	JPVM	JPVM	JPVM	JPVM
		byte	int	byte	int	byte	int
		JDK	JDK	Kaffe	Kaffe	JDK	JDK
Latency	1.2	4.3	4.3	15	18	76	81
Time/byte	.00089	.0013	.0029	.0022	.0022	.0018	.0042

Table 5: Communication Bandwidth for SUN workstation using JDK 1.1.5.

	MB/s				
Bytes	PVM	JavaPVM	JavaPVM	JPVM	JPVM
Sent	byte	byte	int	byte	int
4	.0055	.0013	.00064	.000026	.000026
12	.016	.0029	.0020	.000078	.000076
100	.12	0.020	.018	0.00076	.00063
1000	.62	0.11	.11	0.0085	.0053
10000	1.08	0.47	.32	0.089	.035
100000	1.15	0.77	.38	0.36	.057
1000000	1.15	0.90	.38	0.52	.086

Table 6: Communication Bandwidth for Linux.

Bytes Sent	PVM	JavaPVM byte JDK	JavaPVM int JDK	JPVM byte Kaffe	JPVM int Kaffe	JPVM byte JDK	JPVM int JDK
	MB/s						
4	.0039	.00125	.0008	.00020	.00019	.000052	.000052
12	.012	.029	.0036	.00055	.00055	.00015	.00015
100	.086	.029	.027	.0047	.0048	.0013	.0012
1000	.40	.17	.16	.032	.025	.012	.013
10000	.93	.47	.29	.30	.22	.12	.093
100000	1.1	.72	.34	.52	.51	.39	.19
1000000	—	—	—	.46	.45	.52	.23

greater than the best achieved by PVM.

The latency in JavaPVM and JPVM is the factor that significantly decreases performance. It is not surprising that PVM had the lowest latency of the three. JavaPVM is layered over PVM and was expected to incur some additional overhead due to the wrappers translating between Java and C. However, JPVM's latency was over two orders of magnitude larger than PVM's latency on the SUN workstations. A reduction in the latency in JavaPVM and JPVM would do much to improve the performance of these communication libraries. All the Java runs on the Linux machines had significantly lower latency than Java runs on the SUN workstations. For JPVM the latency on the Linux workstations under Kaffe was less than one sixth of the corresponding value on the SUNs. Table 5 and table 6 show how large latencies significantly reduce the effective bandwidth for small messages. JPVM's effective bandwidth for a 1,000 byte message on Linux is over three larger than the effective bandwidth for the same sized message on the SUN despite the per byte cost being very similar.

The tests on the Linux machines show little difference in sending different primitive types in Kaffe. However, on both JDK 1.1.3 and JDK 1.1.5 a significant penalty was incurred and the integer arrays took approximately twice the time for communication than equivalent size byte arrays. For messages greater than 100,000 bytes sent by JPVM on the SUN the penalty for sending integer arrays was greater than six times the cost of sending the equivalently sized byte array.

The Just-In-Time translation system in Kaffe on the Linux platform significantly reduces the latency in JPVM. As shown in table 4 the Kaffe's latency was computed to be 15ms versus 76ms for JDK 1.1.3 on identical machines. Because Kaffe does not have libraries that support object serialization, the JPVM daemon for the Kaffe runs was still executed under JDK 1.1.3, but this has a relatively small contribution to the overhead.

There are a number of reasons for the lower communication performance of the Java implementations of PVM:

- For JavaPVM the basic communication library is built on the native C PVM library. JavaPVM uses wrapper routines to interface between the Java and the C routines. This introduces two forms of overhead: an additional call to switch between Java and C during runtime and code to reformat arguments for PVM routines.
- The Just-In-Time compilers perform dynamic translation of Java bytecode into native machine language. These translations are performed the first time a function is called and was observed in the amount of time required to send the first message in the tests. Even after the function is translated there is still some overhead each time the function is called to determine whether a native translations already exists.
- All data types in Java other than primitive data types, e.g. scalar integers, floats, and doubles, require memory allocation. As a result, structures of structures require multiple calls to the memory allocation routines to allocate space for each of the fields in the structure. An equivalent C program structure of

structures requires only a single call to the memory allocation routine. Every time memory is allocated in Java, a constructor must be called to initialize it. Thus, the dynamic creation and removal of objects in JPVM can create additional overhead. For very large messages (1,000,000bytes) in Kaffe the overhead of memory managements reduces the effective bandwidth, shown in table 6.

- The way that Java I/O is implemented was found to be a major source of overhead in some benchmarks [10]. JPVM make heavy use of the sockets implemented in the Java I/O library.
- Primitive types other than bytes require additional overhead to send in a portable manner causing the overall performance to be lower. For Kaffe the additional overhead is not noticeable. For JDK which interprets the bytecodes this overhead was significantly more. This effect has been observed by Hiro [11].

This work was performed using a rather low-performance interconnection network, 10Mb/s Ethernet. Even with this rather low performance interconnect the Java communication libraries were unable to saturate the channel. The current Java implementations of communication libraries would derive little benefit from higher performance interconnection networks such as 100Mb/s and Gigabit Ethernet because the interconnection networks are not limiting performance. The software performance must be improved for the limitations in the 10Mb/s Ethernet to become the limiting factor. This observation was also made by Hiro [11].

7 Conclusion

Due to the better communication performance of JavaPVM over JPVM, JavaPVM is currently a better choice for running communication intensive Java applications. Another advantage of JavaPVM over JPVM is its inter-operability with native languages. Thus, even though Java simplifies the implementation of JPVM, the use of the Java interface to the C implementation of PVM is a better strategy for high performance with the current communication library implementation.

However, JPVM is a relatively recent development and it has not been extensively tuned for performance. The code could certainly be optimized for better performance. Performance tuning coupled with the improvements in the performance of Just-In-Time compilers make it quite likely that JPVM can achieve performance comparable to native PVM. The standardization of the Java Virtual Machine makes the JPVM code much simpler to maintain across heterogenous machine than PVM and eliminate the need for code to translate between different machine data formats.

8 Acknowledgements

We would like to thank Adam Ferrari for his assistance in getting JPVM to work on Linux.

References

[1] A. Geist, A. Beguelin, J.Dongarra, W. Jiang, R.Manchek, and V. Sunderam. *PVM: Parallel Virtual Machine - A Users' Guide and Tutorial for Networked Parallel Computing Scientific and Engineering Series.* MIT Press, 1994.

[2] MPI: A Message-Passing Interface Standard. Technical Report us-cs-94-230, Uni. of Tennessee, Knoxville, Tennessee, April 1994.

[3] J. Gosling, B. Joy, and G. Steel. *The Java Language Specification.* Addison-Wesley Publishing Company, Reading, Massachusetts, 1996.

[4] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification.* Addison-Wesley Publishing Company, Reading, Massachusetts, 1997.

[5] C.-H. A. Hsieh, J. C. Gyllenhall, and W.-M. W. Hwu. Java Bytecode to Native Code Translation: The Caffeine Prototype and Preliminary Results. In *Proceedings of the 29th Annual International Symposium on Microarchitecture*, Paris, France, December 1996.

[6] David A. Thurman. JavaPVM: The Java to PVM Interface, December 1996. URL: http://www.isye.gatech.edu/chmsr/JavaPVM/.

[7] Adam J. Ferrari. JPVM: The Java Parallel Virtual Machine, June 1996. URL: http://www.cs.virginia.edu/~ajf2j/jpvm.html.

[8] Adam J. Ferrari. JPVM: Network Parallel Computing in Java. Technical Report CS-97-29, Uni. of Virginia, Charlottesville, Virginia, December 1997.

[9] Jack Dongarra and Tony Hey. The ParkBench Benchmark Collection, November 1995. URL: http://parallel.rz.uni-mannheim.de/top500/reports/report94/benrep3/benrep3.html.

[10] C.-H. A. Hsieh, M. T. Conte, T. L. Johson, J. C. Gyllenhall, and W.-M. W. Hwu. A Study of the Cache and Branch Performance Issues with Running Java on Current Hardware Platforms. In *Proceedings of IEEE CompCon '97*, San Jose, California, February 1997.

[11] S. Hirano, Y. Yasu, and H. Igarashi. Performance Evaluation of Popular Distributed Object Technologies for Java. In *Proceedings of the ACM 1998 Workshop on Java for High-Performance Network Computing*, pages 81–90, Palo Alto, California, February 1998.