

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

February 7, 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) [GBJ⁺94] and the Message Passing Interface (MPI) [MPI94] 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 [GJS96] 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) [LY97], 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 [HGH96]. 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 [Thu96] and JPVM [Fer96, Fer97], 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 [GBJ+94] 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. These interface routines allow the creation and termination of tasks across heterogeneous clusters of computers as well as 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 [Thu96] 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, while C or Fortran could be used to write 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 [Fer96, Fer97] 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. 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 [DH95] 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

Table 1: Round Trip Times

Bytes Sent	PVM(ms)	JavaPVM(ms)	JPVM(ms)
1	.803	4.98	132
100	.880	5.12	131
1000	1.73	8.86	117
10000	9.73	21.3	112
100000	89.6	129	278
1000000	892	1110	1910

Table 2: Cost of sending messages

Bytes Sent	PVM(ms)	JavaPVM(ms)	JPVM(ms)
Latency	.75	10.5	115
Time/byte	.00089	.0011	0.0018

task 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 them to the slave task. The slave task sends back the received message to the master task. The communication times were measured for messages with payloads of 1, 100, 1,000, 10,000, 100,000, and 1,000,000 bytes.

The 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 compiled using SUN's 1.1.5 Java Development Kit and executed on SUN's Java Virtual Machine that uses Just-In-Time(JIT) compilation technology. PVM 3.3 compiled with GCC 2.7.2 was used for the PVM and JavaPVM experiments.

The round-trip times for the ping-pong benchmark are shown in table 1. Table 2 shows the round-trip latency and time per byte calculated by linear regression for each of the communication libraries. Finally, table 3 lists the effective bandwidth for each of the tests.

Table 3: Communication Bandwidth

Bytes Sent	PVM(MB/s)	JavaPVM(MB/s)	JPVM(MB/s)
1	0.0012	0.00020	0.0000076
100	0.114	0.020	0.00076
1000	0.293	0.11	0.0085
10000	1.03	0.47	0.089
100000	1.12	0.77	0.36
1000000	1.12	0.90	0.52

6 Discussion

Table 2 summarizes the costs associated with the communication packages as latency and time per byte sent. This table clearly shows the performance of each of the communication packages. A surprising feature of the table is that the time per byte is within a factor of two between the fastest (PVM) and slowest (JPVM). With the additional overheads in Java, a larger difference in the time per byte was expected.

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. However, JPVM's latency was over two orders of magnitude larger than PVM's latency. A reduction in the latency in JavaPVM and JPVM would do much to improve the performance of these communication libraries. Table 3 shows how large latencies significantly reduce the effective bandwidth for small messages.

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

- For the 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. This translations is performed the first time a function is called. We strived to avoid including this overhead by not including the time for the first iteration. However, there is still some overhead to determine whether the function that is going to be called has been translated into native code.
- 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.
- The way that Java I/O is implemented was found to be a major source of overhead in some benchmarks [HCJ⁺97]. JPVM make heavy use of the sockets implemented in the Java I/O library.

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.

References

- [DH95] Jack Dongarra and Tony Hey. The ParkBench Benchmark Collection, November 1995. URL: <http://parallel.rz.uni-mannheim.de/top500/reports/report94/benrep3/benrep3.html>.

- [Fer96] Adam J. Ferrari. JPVM: The Java Parallel Virtual Machine, June 1996. URL: <http://www.cs.virginia.edu/ajf2j/jpvm.html>.
- [Fer97] JPVM: Network Parallel Computing in Java. Technical Report CS-97-29, Uni. of Virginia, Charlottesville, Virginia, December 1997.
- [GBJ⁺94] 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.
- [GJS96] J. Gosling, B. Joy, and G. Steel. *The Java Language Specification*. Addison-Wesley Publishing Company, Reading, Massachusetts, 1996.
- [HCJ⁺97] C.-H. A. Hsieh, M. T. Conte, T. L. Johnson, 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.
- [HGH96] 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.
- [LY97] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley Publishing Company, Reading, Massachusetts, 1997.
- [MPI94] MPI: A Message-Passing Interface Standard. Technical Report us-cs-94-230, Uni. of Tennessee, Knoxville, Tennessee, April 1994.
- [Thu96] David A. Thurman. JavaPVM: The Java to PVM Interface, December 1996. URL: <http://www.isye.gatech.edu/chmsr/JavaPVM/>.