Message-Passing Computing with Java: Performance Evaluation and Comparisons

Vladimir Getov[1], Quanming Lu[2], Marry Thomas[3], Matthew Williams[1]
[1] School of Computer Science, University of Westminster, Watford Road, London, UK
(e-mail: {getovv, williams}@wmin.ac.uk)
[2] Institute of Information Sciences and Electronics, University of Tsukuba, Japan
(e-mail: qmlu@is.tsukuba.ac.jp)
[3] Scientific Computing Department, San Diego Supercomputer Center, La Jolla, CA, USA
(e-mail: mthomas@sdsc.edu)

Abstract

The development of Java has seen increasing attention as the most popular platform for distributed computing. However, despite Java's advantages in the area of portability and rapid prototyping, its efficiency is unavoidably compromised through its commitment to portability. In this paper we present performance analysis and comparisons of evaluation results for both Java and C/Fortran on three different message-passing parallel platforms - a shared memory multi-processor (Sun E4000), a Linux cluster, and a distributed memory computer (IBM SP-2). The NAS Embarrassingly Parallel and Integer Sort benchmarks were selected for this evaluation. Both the original Fortran/C codes and Java versions of these two kernels were used for obtaining the performance measurements as part of our project. The evaluation results demonstrate the feasibility of message-passing computing with Java on a wide range of computer platforms. Depending on the system and the software components installed, significant impact on the message-passing performance will have the efficiency of the native MPI library and the version of the Java platform.

1. Introduction

The computer platforms suitable for achieving high performance have generally been thought of as in the realm of "supercomputers". Such high-end platforms usually include fast vector computers, large shared memory machines, or distributed memory multi-processor systems with high-speed interconnects. Cluster computers and massively parallel processing systems have the same basic distributed memory parallel architecture, but clusters generally have slower communication interconnects between nodes. Recent advances in networking technology, however, make clusters a more feasible option for tackling the problems traditionally dealt by supercomputers. Of course, in order to turn a network of workstations into something that behaves more like a supercomputer additional message-passing software must be layered on top of the conventional communication sub-system.

Actually, local area networks may not be the only communication sub-systems suitable for the introduction of faster interconnects. The growth of the Internet offers the world of high performance computing massive computational power at very little cost. Programs may be written to take advantage of resources based in logically and geographically different locations without change to their existing infrastructure. Problems arise, however, because of the diversity of the operating systems, CPU's and networks involved. To overcome these problems a paradigm must be created which makes the heterogeneous environment opaque to the programmer. Ideally, the programmer may want to create a software application which could run on a cross-platform metacomputing environment. In this case, the environment may be comprised of multi-processor shared memory machines or a network of workstations or both. Efforts to meet these ends must not only provide a robust framework within which to work but must also balance preservation of efficiency.

The development of Java has seen the above possibility brought a step closer – its platform independent bytecode representation is ideal for distributed computation. However, despite Java's advantages in the area of portability and rapid prototyping the scientific community have been reluctant to embrace Java. Its efficiency is unavoidably compromised through its commitment to portability. This makes Java unattractive as a programming language for achieving high performance.

1066-6192/01 $10.00 © 2001 IEEE

Over the last four years supporters of the Java Grande Forum [7] have been working actively to address some of the issues involved in overcoming this obstacle. The goal of the forum has been to develop consensus and recommendations on possible enhancements to the Java language and associated Java standards, for large-scale ("Grande") applications. There have been many active projects underway whose aim is to analyze the feasibility of Java for computational tasks associated with High Performance Computing (HPC).

One focus which has arisen from these pursuits has been the concentrated effort toward bringing together Java and the widely accepted Message Passing Interface (MPI) [11]. Within this endeavor are the developments of MPI-like Application Programmer Interfaces (API) for Java that currently include mpiJava [3], JavaMPI [10], and MPIJ [8]. MPIJ is a pure Java implementation, while JavaMPI and mpiJava use the Java Native Interface (JNI) [9] to access the local native MPI libraries and emulate the functionality of standard MPI.

Even if Java does not produce the levels of performance needed for those parts of Grande problems that are computationally intensive and need to run on an HPC system, it can still play a major role in the distributed and metacomputing aspects of these large projects. Thus, it is important to push the development of this language as far as is possible. In this paper we provide recent performance evaluation and comparisons results for different message-passing platforms with Java.

2. Evaluation codes

The NAS parallel benchmarks are well understood, written in Fortran or C, and are most often used to characterize the performance of parallel HPC systems [1]. The Embarrassingly Parallel (EP) and the Integer Sort (IS) NAS parallel benchmarks were used in our performance evaluation. The IS routine evaluates integer operations and bidirectional communications (the sorted keys are exchanged between nodes), while the EP kernel tests floating point operations performance but requires minimal communications. The NAS version of IS is written in C, while the EP code is in Fortran.

Porting code from Fortran or C to Java is not straightforward. In porting the EP (Fortran) and IS (C) NAS benchmarks to Java, every attempt was made to follow the logic, design, and program flow to guarantee that the code tests the same computational features as the original versions. Since these experiments are designed to compare Java to conventional languages, the focus here is on optimization, rather then on object oriented programming design techniques. Although not mutually exclusive, we found that the class and method access costs were significant, and as a result, the Java class designs were kept to a simple set. Note that although there are a few tools available that automatically convert Fortran or C code to Java, these were not used because they are not well understood and do not have optimization features for parallelizing code.

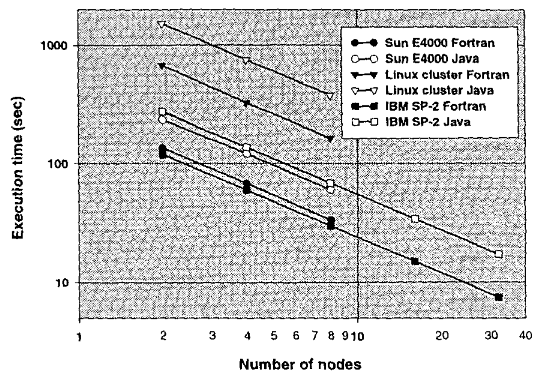
Java, Fortran and C have some similar constructs such as loops, variable declarations and conditional blocks. But there are some key differences as well. For example, parameters are passed by value in Java, but are passed by reference in Fortran, and can also be passed by reference in C when using pointers. The only mechanism open in Java to address this difference is to pass an object that contains a variable that can be changed. The problem here is that there is added overhead when passing an object as compared to passing a basic datatype. However, this is unavoidable. These objects tend to be simple container classes with `public` variables so that the modification can be done without the overhead of calling a method.

There is no Java equivalent to the `include` statements used in C and Fortran or the `common` blocks in Fortran, so array declarations and critical sections of the code were kept inside a single class. Local methods were used in place of subroutine or function calls. The classes were designed so that the key blocks of MPI codes could be identified and replaced by other test MPI libraries. In addition, we found that by defining methods and variables internal to the class as `static`, `private`, or `final` improves performance. This is because the compiler can statically resolve methods at compile time, avoiding the overhead of finding and loading the method at run time.

3. Message-Passing Environment

For our parallel message-passing tests, we chose to use the JavaMPI environment, which is based on wrapping up the Local Area Multicomputer (LAM) MPI library developed in C at the Ohio Supercomputing Center [2]. The purpose for the development of the JavaMPI binding has been to provide Java programmers with the traditional functionality of MPI through a Java interface to legacy MPI libraries. The JavaMPI wrappers to LAM were created using JCI, the Java-to-C interface generating tool [5]. This tool enables communication to the underlying LAM MPI library by using the JNI API [9]. It allows Java code that runs inside the Java Virtual Machine (JVM) to interoperate with native, system-dependent code written in other languages. This JNI binding of a native MPI library to Java offers several immediate advantages, such as the rapid software development associated with Java and the established performance of existing MPI implementations.

Creation of a native library from scratch allows the programmer to implement the native functions with the required JNI structure. This means that the functions may be called directly from within Java code without any additional interfacing. This may be the case to perform some machine specific task not possible from within the JVM. However, the calling of an existing library function requires an additional interface layer. This additional interface layer is known as the wrapper or binding and is necessary to perform the required type conversions before the actual call to the legacy library function is made. Another reason that the JavaMPI wrapper is required is that the structure or prototypes of the MPI library functions would not be compatible with the JNI. A pointer to the run-time environment and a pointer to the Java object executing the native code must be passed at each function call, these are used to make access to data and methods inside the JVM. The JavaMPI function call does not need to include these parameters and so the binding closely follows the MPI 1.0 standard. It is the JNI which inserts these parameters to every native function call. JavaMPI uses these pointers to extract the necessary data required for the MPI function call, and then to update the necessary data stores with the values returned from the actual MPI function call.



No.	Sun E4000		Linux cluster		IBM SP-2	
	F-77	Java	F-77	Java	F-77	Java
2	135.4	236.0	673.7	1514	119.3	274.3
4	68.14	121.4	327.4	755.8	59.70	137.4
8	33.52	59.95	163.8	377.9	29.80	68.60
16	n.a.	n.a.	n.a.	n.a.	15.00	34.30
32	n.a.	n.a.	n.a.	n.a.	7.50	17.20

Figure 1. Execution time vs. number of nodes for the NAS EP benchmark – class A

4. Experimental Results

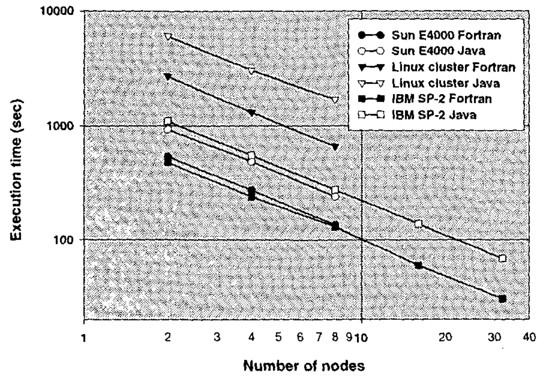
A series of experiments were conducted with both the IS and the EP NAS parallel benchmarks in order to evaluate and compare the performance achievable on three different platforms. The kernels were run in two versions – first when using the standard codes in C or Fortran with the corresponding MPI libraries and then the Java translations of these kernels with the JavaMPI bindings to the same MPI libraries. A distributed memory parallel computer such as the IBM SP-2, a shared memory multi-processor – Sun E4000, and a Linux cluster were selected for our experiments as they cover relatively well the variety of currently available message-passing parallel platforms.

The JVM and the Java compiler used on the IBM SP-2 machine were part of the JDK for AIX, version 1.1.6. The execution environment consisted of IBM's Parallel Operating Environment (POE), which supports the loading and execution of parallel processes across the nodes of the IBM SP2. The machine is built of thin nodes POWER2 Super Chip (P2SC) processors with 256 Mbytes of memory on each processor. The communication subsystem of the SP2 features a high-performance switch which was used throughout the experiments. The NAS EP and IS benchmarks were also run on a 200 MHz dual Pentium Pro processor cluster running Linux Red Hat 6.0 on a 10baseT Ethernet. The same experiments were performed on a 14x336 MHz Ultra Sparc II processor Sun E4000 running Solaris 2.6.

The LAM MPI library was used on the Linux cluster, whilst both the SP-2 and the E4000 provided native MPI libraries for message passing. The JavaMPI wrapper software, created by the JCI tool, was used for the Java bindings to these libraries. The original NAS C and Fortran codes were run on top of the corresponding MPI libraries, while the Java kernels were run on top of the JavaMPI binding to these libraries. Later versions of Java 1.1.x were installed on all platforms.

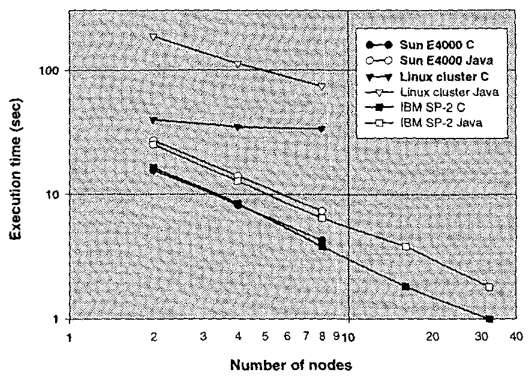
The NAS parallel benchmarks have several specified problem sizes called "classes" in order to ensure comparative results across different platforms and environments. In our study, we have completed experiments for two problem sizes of the EP kernel (class A and class B) and one problem size for the IS benchmark (class A). The corresponding problem sizes in data points for the EP code are 2^{28} for class A and 2^{30} for class B, while the class A problem size for the IS code corresponds to 2^{23} data points [1].

The evaluation results for the EP kernel (class A) are shown in Figure 1. The execution time statistics for class B of the same kernel (Figure 2) does not show any significant differences as far as the relative performance is concerned. In both cases the standard Fortran code using native message-passing performs best on the IBM SP-2. The results on the Sun E4000 are slightly slower while the Linux cluster delivers timings nearly a magnitude behind the other two platforms. The programs run approximately 2.5 times slower in Java than their corresponding Fortran counter parts. In both cases, however, the codes demonstrate good



No.	Sun E4000		Linux cluster		IBM SP-2	
	F-77	Java	F-77	Java	F-77	Java
2	535.4	931.3	2694	6056	477.3	1099
4	272.7	486.1	1310	3029	238.6	548.2
8	134.0	239.2	654.7	1691	129.3	274.3
16	n.a.	n.a.	n.a.	n.a.	59.70	137.3
32	n.a.	n.a.	n.a.	n.a.	30.40	68.70

Figure 2. Execution time vs. number of nodes for the NAS EP benchmark – class B



No.	Sun E4000		Linux cluster		IBM SP-2	
	C	Java	C	Java	C	Java
2	15.86	27.45	39.98	185.3	16.60	25.40
4	8.30	14.16	35.20	112.8	8.50	12.80
8	4.26	7.35	33.81	74.24	3.80	6.40
16	n.a.	n.a.	n.a.	n.a.	1.80	3.80
32	n.a.	n.a.	n.a.	n.a.	1.00	1.80

Figure 3. Execution time vs. number of nodes for the NAS IS benchmark – class A

scalability within the range allowed by the hardware configurations.

The Java implementation on the Sun Enterprise 4000 machine clearly outperforms the corresponding implementation for AIX on the IBM SP-2 for those two experiments. Of course, this can only be attributed to the specific versions of the Java software installed on the two machines. Newer software versions will almost certainly deliver different comparative performance results. Therefore, separate measurements should be taken if necessary in each specific case in order to evaluate this particular issue.

The benchmarking results obtained with the IS kernel (class A) are shown in Figure 3. The IS code is a relatively stronger test for the message-passing environment involving a number of bi-directional communications. This changes the computation/communication balance and therefore reduces the overhead introduced by Java for both the IBM and the Sun machines. For the Linux cluster, however, the more intensive communications appear to be a problem particularly when running the standard C code. The results for different numbers of nodes are almost the same, because of the relatively slow 10baseT Ethernet which obviously becomes a performance bottleneck in this case.

5. Discussion

The data reflected therein bring to bear several questions which need to be addressed. Foremost is the need to identify the sources of the performance penalty of using the message passing paradigm from Java in comparison to traditional message passing programming in C or Fortran. Therefore, we have also focused on the performance penalty introduced by the respective components of a JNI-Wrapped MPI process. In such a process, there is a layer where execution of the Java bytecode takes place (in the JVM); a layer where execution is entirely done inside of the MPI library; and another JNI (C-based) layer which permits the transition of execution to and from the Java bytecode into the MPI library calls. Further investigation into the breakdown of time spent in each of these layers is needed in order to understand the sources of the performance penalty of using the JavaMPI environment.

The performance penalty introduced by the MPI libraries is relatively very well studied and understood. In order to gain a more detailed insight, we have instrumented the JavaMPI binding, and gathered additional time measurements. It turns out that the cumulative time spent in the wrapper software layer is typically within $1\mu s$ in all cases, and thus has a negligible share in the breakdown of the total execution time. Therefore, the significant performance difference in comparison to the corresponding C or Fortran results should be attributed to JNI, and the extra data copies in particular [6]. A detailed study of the JVM performance and more precisely how the parameters of the JVM (heap size, stack size, garbage collection mode, just-in-time compilation, etc.) influence the performance on platforms such as the ones used in our study is obviously a very interesting one but beyond the scope of this paper.

6. Conclusion

The Message-Passing Working Group of the Java Grande Forum was formed in the second half of 1998 as a response to the organized collaborations within the Java Grande Forum to develop a single MPI-like API specification for the Java programming language. An immediate goal was to discuss a common API for MPI-like Java libraries and an initial draft was distributed at Supercomputing'98. In addition to faster JVM/JNI implementations, the communication performance improvements very much depend on the efficient design of an MPI-like API in Java. This has been an area of active research and development during the last couple of years for both wrappers to existing MPI libraries and pure Java designs. To avoid confusion with standards published by the original MPI Forum the nascent API specification is called MPJ (Message Passing for Java) [4].

The MPJ specification has been designed to work equally well for both wrapper and pure Java implementations. In general, our evaluation results in this study should be also valid for wrapper versions of MPJ despite the fact that JavaMPI does not conform to the specification document. Depending on the platform and the software components installed, significant impact on the message-passing performance will have the efficiency of the native MPI library and the version of the Java platform, including JNI. For pure Java implementations of MPJ, however, a separate evaluation exercise will have to be conducted.

The work presented in this paper has helped to identify the usefulness of message-passing in Java for real applications while bringing to light the aspects of Java's performance which need to be improved upon in order to realize high-performance communication. The performance measurements which have been completed under this project go directly to the questions which arise concerning Java's future role in scientific and high-performance computing.

7. Acknowledgements

The authors would like to thank the San Diego Supercomputer Center and the University of Wales - Cardiff for the use of their computer systems (IBM SP-2 and Sun E4000 respectively) and for the technical support that they provided for this research.

References

[1] D. Bailey, E. Barszcz, J. Barton, D. Browning, R. Carter, L. Dagum, R. Fatoohi, S. Fineberg, P. Frederickson, T. Lasinski, R. Schreiber, H. Simon, V. Venkaakrihnan, and S. Weeratunga. The NAS parallel benchmarks. Technical Report RNR-94-007, NASA Ames Research Center, 1994. http://science.nas.nasa.gov/Software/NPB/.

[2] G. Burns, R. Daoud, and J. Vaigl. LAM: An open cluster environment for MPI. In *Proceedings of Supercomputing Symposium '94*, Toronto, Canada, 1994.

[3] B. Carpenter, G. Fox, G. Zhang, and X. Li. A draft Java binding for MPI. November 1997. http://www.npac.syr.edu/projects/pcrc/HPJava/mpiJava.html.

[4] B. Carpenter, V. Getov, G. Judd, A. Skjellum, and G. Fox. MPJ: MPI-like message passing for Java. *Concurrency: Practice and Experience*, 12, 2000 (to appear).

[5] V. Getov, S. Flynn-Hummel, and S. Mintchev. High-performance parallel programming in Java: Exploiting native libraries. *Concurrency: Practice and Experience*, 10(11-13):863–872, 1998.

[6] R. Gordon. *Essential JNI: Java Native Interface*. Prentice Hall, 1998.

[7] Java Grande Forum. Making Java work for high-end computing. Technical Report JGF-TR-1, November 1998. http://www.javagrande.org/reports.htm.

[8] G. Judd, M. Clement, and Q. Snell. Dogma: Distributed object group metacomputing architecture. *Concurrency: Practice and Experience*, 10(11-13):977–983, 1998. http://ccc.cs.byu.edu/DOGMA/.

[9] S. Liang. *the Java Native Interface: Programmer's Guide and Specification*. Addison Wesley, 1999.

[10] S. Mintchev and V. Getov. Towards portable message passing in Java: Binding MPI. In: M. Bubak, J. Dongarra, and J. Wasniewski (Eds.). *Recent Advances in PVM and MPI, LNCS*, 1332:135–142, Springer, 1997. http://perun.hscs.wmin.ac.uk/JavaMPI/.

[11] MPI Forum. MPI: A message-passing interface standard. *International Journal of Supercomputer Applications*, 8(3/4), 1994.