Benchmarking Java on the IBM SP-2

Mary Thomas\textsuperscript{1}, Vladimir Getov\textsuperscript{2}, Matthew Williams\textsuperscript{2}, Roger Whitney\textsuperscript{3}

\textsuperscript{1}Scientific Computing Department, San Diego Supercomputer Center, USA. (e-mail:mthomas@sdsc.edu, http://www.sdsc.edu/~mthomas)
\textsuperscript{2}Distributed and High Performance Computing Group, University of Westminster, Watford Road, London, UK.  
(e-mail: {v.s.getov, williams}@wmin.ac.uk, http://perun.hscs.wmin.ac.uk)
\textsuperscript{3}Department of Mathematical and Computer Science, San Diego State University, San Diego, California, USA  
(e-mail: whitney@cs.sdsu.edu, http://www.eli.sdsu.edu)

Abstract

In this paper we present results and analysis of benchmark tests that directly compare the performance of JavaMPI, Fortran and C routines on the 128-node IBM SP-2 HPC system, at the San Diego Supercomputer Center. The suite of tests include scalability studies of the NAS Embarrassingly Parallel and Integer Sort Benchmarks for 2 to 128 processors, along with results from basic operations and communication measurements. We present results from a set of Java classes that were developed to implement these two kernels, and classes that measure the costs of some basic Java operations. The Java versions of the NAS benchmark results presented here are based on IBM's JDK 1.1.6 for AIX (with JIT enabled) and the JavaMPI libraries that run on the LAM MPI Libraries. These classes have been written generally enough such that other MPI-like Java implementations may be easily incorporated in future studies. For comparison, we have completed experiments for three different versions of our programming environment - the original NAS codes running on the IBM POE/MPI libraries, the original NAS codes running on the LAM MPI libraries, and our Java versions using JavaMPI on top of LAM.

1 Introduction

Because Java is capable of cross platform operation with the same binaries (byte) code, and it is a threads based language, it appears to be well suited as a programming environment for the next generation of distributed and parallel computing applications. If the performance of Java compilers and virtual machines (VM) increases to a level that is comparable to the current standards of Fortran and C, then Java could also be considered as a candidate for high-performance computing (HPC) [1]. Even if Java does not produce the levels of performance needed for HPC computations, it can still play a major role in the distributed and metacomputing aspects of these large applications. Thus it is important to characterize the performance of Java against known parallel programming languages such as C and Fortran in order to determine where Java can be most useful, and to indicate which areas require improvements.

Parallel libraries such as the Message Passing Interface (MPI) and Parallel Virtual Machine (PVM) can be implemented in any language (Fortran, C, or Java) and are the mainstream libraries for HPC tasks [2,3]. For this paper, we consider only the MPI libraries, although there is ongoing research on the Java-PVM model, such as JPVM [4] and JavaNOW (Network of Workstations) [5]. Current Java implementations of the MPI library include mpiJava [6], JavaMPI [7], and JMPI [8]. JMPI is pure message passing Java, while JavaMPI and MPIJava use the Java Native Interface (JNI) to access the local native MPI libraries and emulate the functionality of standard MPI.
Previous work on the SP, based on JavaMPI and LAM libraries indicate that for the NAS Integer Sort (IS) benchmark, Java runtimes are about 2.5 times slower than the C version, and are comparable to the C version when the IBM hpcj compiler is used [9]. These results are encouraging for such a new language, and we felt that we could expand on the JavaMPI studies to investigate the scalability of parallel Java libraries to larger systems. For these experiments, we chose to work with the NAS parallel benchmarks because they are well known and understood to the HPC community and because we wanted to do a direct comparison of JavaMPI to standard HPC languages [10].

In this paper we present the results of our study, which was performed on a 128 node IBM SP-2 system, for the Embarrassingly Parallel (EP) and Integer Sort (IS) benchmarks. These test exercise floating point calculations and integer communications within the Java parallel environment. Section 2 of this paper introduces and defines the evaluation codes used in this study, while Section 3 presents the results. The discussion of these results and our conclusions are presented in Sections 4 and 5. The compete results, and source code, can be found in the thesis work done by the first author of this paper [11], which is available on-line.

2 Evaluation Codes

The NAS parallel benchmarks are well understood, written in C or Fortran, and are often used to characterize most HPC systems. The IS routine tests integer operations and communications, while the EP codes test floating point operations and has minimal communications. For our parallel, MPI Java tests, we chose to use the JavaMPI library, which is based on the C-based Local Area Multicomputer (LAM), MPI library developed at the Ohio Supercomputing Center [12]. The JavaMPI wrappers to LAM were created using JCI, the Java-2-C interface generating tool [7]. This tool enables communication to the underlying LAM MPI library by using the Java NMI/JNI1.0 API to gain access to native C code generated by JCI.

The LAM libraries are free, and there are ports to many common scientific research platforms (Solarix, AIX, Unix, etc.), which makes JavaMPI extremely portable. However, LAM uses the TCP/IP protocol, which is not the fastest communication mechanism on the SP which uses the proprietary high speed interconnect or user-space nodes. These nodes can only be accessed when running code under the IBM parallel operating environment (POE). This added complexity to the benchmarking experiments: the study required a characterization of codes performance under three environments: POE, LAM, and JavaMPI+LAM. In addition, we included characterization of message passing and communication in order to account for costs associated with using the LAM-MPI libraries, these will affect the JavaMPI results.

Additionally, as part of the calibration of the benchmark results, we developed test classes that measure basic operations. These tests are important because both the EP and IS benchmarks access several mathematical libraries, and it is well known that these can have high overhead costs associated with them. Each of the evaluation codes is describe below.

2.1 Communication

The best performance results on the IBM SP will be obtained for programs using the high-performance MPI libraries that run under the IBM Parallel Operating Environment (POE) environment, which has access to specialize hardware. Fortran and C programs that are compiled using the xlc, xlC, and xlf90 compilers will be able to access the faster communication hardware. Since LAM is based on the TCP/IP protocol, we expect that all LAM programs will have a reduced performance entirely due to the performance of the LAM environment. Note that the LAM-based C and Fortran routines were not compiled.
using the xlc and xlf90 compilers because of compilation errors that occurred when trying to link in the LAM MPI library.

The IS routine tests integer operations and bi-directional communications (the sorted keys are exchanged between nodes), while the EP tests have minimal communications. Tests to characterize message-passing behavior for sending and receiving integers were done using a bi-directional ping algorithm: each processor P sends a message of length n to the other P-1 processors, and then waits to receive the messages sent back from the other processors. Any costs associated with either message startup (latency) or communication (bandwidth) will show up under these test conditions. The NAS version of IS is written in C, so the bi-directional ping routine was written in C and JavaMPI. The C version is compiled against the POE and LAM MPI libraries.

2.2 Basic Operations

The focus of this study is to benchmark parallel Java using an MPI based library on a large parallel system. The basic operations test suite is included in order to measure the overhead (if any) associated with using either the LAM libraries or the JavaMPI libraries. These tests are designed to isolate effects, for example, that might be due to the JNI or to some unknown advantage gained by compiling with xlc. In addition, these tests were used as optimization aids.

A set of modules and classes were developed to measure the cost of doing basic arithmetic operations (+, -, *, /) and math library accesses. These serial routines were implemented in both Java and Fortran. For the Java classes, a flag was added to test for class and variable scope that switched between accessing local variables (within the method), class variables, or external variables. All timings were based on the MPI function MPI_Wtime(). These tests were designed to examine broad characteristics and to support the analysis and comparison between the LAM/POE/JavaMPI parallel environments, and the Fortran/C/Java routines, and were not intended to benchmark the SP-2.

2.3 NAS Kernels

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 native languages, the focus here is on optimization, not object oriented 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. Parameters are passed by value in Java, but are passed by reference in Fortran by default, and can be passed by reference in C by using pointers. The only mechanism open in Java 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 or COMMON blocks used in C or 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.

2.4 Timing Routines

A suite of timing classes and routines were developed to analyze all test results. These routines include statistical analysis modules and data output formatting routines. These routines allow us to place timing calls around critical block sections. This methodology is similar to that used in the NAS benchmarks.

All timings were done using the MPI function, MPI_Wtime(), which was first calibrated to serial timers. Both the POE and LAM serial timers were in agreement with the timings obtained from MPI_Wtime(). A Java class was written to calibrate MPI_Wtime() against the serial timer, System.currentTimeMillis. The resultant timings were in agreement to within 0.0055% for long timing tests similar to those that were run in the NAS benchmarks (tens of seconds).

3. Results

Because of the complexity introduced by the use of the LAM MPI libraries, where appropriate, all programs were compiled in either Fortran77 or C, and a similar version was compiled using Java (IBM JDK1.1.6 + JIT, with -O optimization flag). Each test was run 3 ways: first, the programs were run under the IBM Parallel Operating Environment (POE) using the high-performance MPI library; then the tests were repeated using the LAM MPI library (built on C), and run under the LAM parallel environment; finally, the same tests were run using the JavaMPI libraries that run on top of the LAM parallel environment. For the rest of this paper, to distinguish language implementations, the following notation is used for Fortran, C, and Java programs, based on which MPI library is used: f77-POE, f77-LAM, C-POE, C-LAM, and JavaMPI.

A note on the compilation of these routines. For all codes, the f77-POE and C-POE programs were compiled using the xlc, xlC, and xlf90 compilers and have access to the IBM POE MPI libraries. The f77-LAM and C-LAM versions were not compiled using the xlc and xlf90 compilers because of compilation errors that occurred when trying to link in the LAM MPI library. Rather, we used the compiler wrapper scripts provided by the LAM developers, hcc and hf77, which access native C and Fortran compilers.

All tests were run in the batch mode, i.e., the jobs were submitted to the queue. This made it difficult to get fast turn-around times, and did not facilitate debugging. In addition, due to the high machine usage of the SP system, it was difficult to get 128 node data.

3.1 Communications

For the communication measurements, the ring program was run using the 3 MPI libraries: IBM Parallel Operating Environment (POE), the Local Area Multicomputer (LAM), and JavaMPI. Tests were run for 2, 4, 8, 16, 32, and 64 processors and for up to 64 MByte packet lengths. Table 1 summarizes the results for the latency (message start up time) and bandwidth, averaged over all processor runs. Note that
Table 1: Bandwidth measurements of the three MPI libraries on the IBM SP-2.

<table>
<thead>
<tr>
<th>MPI Library</th>
<th>C-POE</th>
<th>C-LAM</th>
<th>JavaMPI</th>
<th>us</th>
<th>ip</th>
</tr>
</thead>
<tbody>
<tr>
<td>Message Startup Time</td>
<td>38.03</td>
<td>189.8</td>
<td>197.8</td>
<td>40</td>
<td>300</td>
</tr>
<tr>
<td>Saturation Bandwidth</td>
<td>106.26</td>
<td>68.57</td>
<td>67.66</td>
<td>100</td>
<td>40</td>
</tr>
</tbody>
</table>

in this table, the last two columns represent values for the IBM SP-2 for both the standard TCP/IP protocol (ip), and the proprietary user space (us) protocol developed by IBM.

The saturation bandwidth measured for the C-POE ring test was 106.26 Mbytes/sec, which agrees with the published values for the high speed interconnect switch on the IBM SP-2 systems [13,14]. For the case of C-LAM and JavaMPI, both behave quite similarly, but the saturation bandwidth limits for both are roughly 65% of the C-POE implementation. These numbers agree with the published values for the TCP/IP thin-node protocol, where the latency and bandwidths are around 200 micro-sec and 60 MB/sec, respectively [13].

The JavaMPI bandwidths match those of LAM, which indicates that JavaMPI has minimal overhead associated with sending and receiving messages. Although this cannot be fully determined until a non-LAM based implementation is installed, it would appear that if Java libraries could be run on top of the native IBM MPI library, then the startup times would be reduced and the message passing bandwidth overhead would be reduced.

### 3.2 Basic operations

Performance measurement and testing of basic operations on HPC systems is non-trivial - hardware is complex and a smart compiler can optimize away your test loops if they are too simple. The intent of the tests run here is to examine the behavior of code under the parallel environments that are a part of this study, not to benchmark basic operations on the SP system (those values are known and published in

<table>
<thead>
<tr>
<th>DataType</th>
<th>OpCode</th>
<th>f77-POE</th>
<th>f77-LAM</th>
<th>JavaMPI</th>
</tr>
</thead>
<tbody>
<tr>
<td>integer/int</td>
<td>Add</td>
<td>3.35E-09</td>
<td>3.47E-09</td>
<td>2.52E-08</td>
</tr>
<tr>
<td>integer/int</td>
<td>Mul</td>
<td>1.23E-08</td>
<td>1.26E-08</td>
<td>2.51E-08</td>
</tr>
<tr>
<td>integer/int</td>
<td>Div</td>
<td>1.21E-07</td>
<td>1.23E-07</td>
<td>2.20E-08</td>
</tr>
<tr>
<td>integer*8/long</td>
<td>Add</td>
<td>1.44E-08</td>
<td>3.43E-09</td>
<td>2.51E-08</td>
</tr>
<tr>
<td>integer*8/long</td>
<td>Mul</td>
<td>1.01E-07</td>
<td>1.27E-08</td>
<td>2.51E-08</td>
</tr>
<tr>
<td>integer*8/long</td>
<td>Div</td>
<td>2.17E-07</td>
<td>1.24E-07</td>
<td>2.20E-08</td>
</tr>
<tr>
<td>real/float</td>
<td>Add</td>
<td>7.20E-09</td>
<td>7.41E-09</td>
<td>2.52E-08</td>
</tr>
<tr>
<td>real/float</td>
<td>Mul</td>
<td>1.26E-08</td>
<td>1.29E-08</td>
<td>2.51E-08</td>
</tr>
<tr>
<td>real/float</td>
<td>Div</td>
<td>1.18E-08</td>
<td>1.21E-08</td>
<td>2.20E-08</td>
</tr>
<tr>
<td>double precision/double</td>
<td>Add</td>
<td>3.61E-09</td>
<td>3.69E-09</td>
<td>2.53E-08</td>
</tr>
<tr>
<td>double precision/double</td>
<td>Mul</td>
<td>7.89E-09</td>
<td>8.06E-09</td>
<td>2.51E-08</td>
</tr>
<tr>
<td>double precision/double</td>
<td>Div</td>
<td>8.27E-09</td>
<td>8.48E-09</td>
<td>2.20E-08</td>
</tr>
<tr>
<td>math</td>
<td>Log</td>
<td>2.60E-07</td>
<td>2.64E-07</td>
<td>6.45E-07</td>
</tr>
<tr>
<td>math</td>
<td>Max</td>
<td>8.11E-09</td>
<td>2.52E-08</td>
<td>3.14E-07</td>
</tr>
<tr>
<td>math</td>
<td>Sqr</td>
<td>9.73E-08</td>
<td>9.94E-08</td>
<td>4.41E-07</td>
</tr>
</tbody>
</table>

Table 2: Arithmetic operations costs (seconds/operations) for local variable scope and comparison of Basic Operations costs tests run under the POE, LAM and JavaMPI environments.
Table 3: Summary of NAS EP (Class A) and IS (Classes A, B) Kernels, for up to 128 processors on the IBM SP-2 (166 MHz).

The results are listed in Table 2. Although these are coarse measurements, there are some useful trends and observations that can be made here. With the exception of the integer*8 operations, the performance of the f77-LAM code on the basic operations (+, -, /, *) is close to but not equal to that of the f77-POE code (97%-98%). Both versions have improved timings for floating point compared to integer operations, and both approach the theoretical peak performance of 640 Mflops quoted for the SP-2.

There is a small, but consistent cost associated with using LAM, which will affect the Java measurements, which was not expected. This may be due to the fact that when installing and compiling LAM, there are automated scripts provided by the developer that are used to compile and run programs for LAM. These scripts may not be optimized for the SP-2/AIX environment, which could affect performance. The JavaMPI timings are nearly the same for all operations, and in general cost more than the native code for most cases. The increased cost and consistent timing is most likely due to the overhead associated with calls to the JNI. In addition, there is a noticeable cost associated with the mathematical routines Log(), Max(), and SQRT(). These routines were chosen because they are used in the EP benchmark. In general,
then, we can expect to observe reduced performance of JavaMPI compared to LAM due to these basic operation costs.

### 3.3 NAS Kernels

The EP and IS benchmark kernels were run for classes A, B, and C, on the SP for the f77-POE, f77-LAM and JavaMPI libraries, on 2 to 128 nodes (memory permitting). The classes represent different problem sizes and increase with letter. For the EP benchmark, which exercises floating point calculations, Class sizes A, B, and C are \(2^{28}, 2^{30}, \) and \(2^{32}\), respectively. The IS benchmarks, which exercise integer communication, for Class sizes A, B, and C the number of keys to be sorted is \(2^{23}, 2^{25}, \) and \(2^{27}\), respectively. The results for EP/Class C and IS/Classes A and B are summarized in Table 3, for the f77-POE, C-LAM, and JavaMPI parallel environments.

All class sizes (A, B, and C) were run on \(P = 2, 4, 8, 16, 32, 64,\) and 128 processors, for the EP benchmark. For the IS kernel, Classes A, B, and C were run on 2, 4, 8, 16, 32, and 64, and 128 nodes, with the exception of the Class C integer sort case. In this case, the size of the job was too large for the 256 MWord nodes on the SP, so there is no data for 2, 4, or 8 processors. Because the 128 node cases did not run successfully, there is only data for 16, 32, and 64 processors. For the f77-POE reference data, the internally published NPACI NAS benchmark data [14]. These results are maintained for the SP and agree well with the published NAS benchmark data [10], and there is no need to reproduce these cases.

For the EP kernel, the f77-POE, f77-LAM and JavaMPI, the results scale nearly linearly with the number of processors. Figures 1 and 2 plot the dependency of the runtime and floating-point operations per second (Mop/s) for 2 to 128 processors. These plots are for the Class-C case, for all three MPI libraries, and the behavior for Classes A and B are similar. In these figures, the f77-POE, f77-LAM and JavaMPI data are represented by the symbols shown in the table on the right hand side: the f77-POE data is represented by a solid diamond, the f77-LAM data is represented by an open square, and the JavaMPI data is represented by an open triangle. The f77-LAM timings are about 86% of the f77-POE rate. The JavaMPI results are within 50% of the f77-LAM
timings, and 44% As expected, there is overhead associated with using the LAM libraries, but it is greater than the 3% performance costs measured in the basic operations tests: the f77-LAM Mflop/s rate of the f77-POE data.

Figures 3, 4, and Figures 5, 6 show results for the runtimes and floating point operations per second, as a function of processor number, for the Class-A and Class-B IS kernels, respectively. For the IS case, neither C-LAM nor JavaMPI scale as well as the C-POE code, and the performance begins to roll off for the 64 and 128 node cases. Due to time constraints and machine availability (the SP at SDSC is a heavily used system), we were only able to obtain 128 node runs, for all 3 parallel environments, for the Class-B case. The C-LAM performance is again within about 85-86% of the C-POE run-times and Mop/s, while the JavaMPI performance is within about 75% of C-LAM results.
The JavaMPI IS data closely follow the C-POE curves out to 32 nodes, and then a performance rollover begins to occur for both C-LAM and JavaMPI. After several tests and experiments, the measured performance for the Java Class A, P=64 node runs was improved by increasing the memory allocated to the program by the JVM. This was done by passing memory arguments to the JVM: \texttt{java -ms1000m -mx1000m}, where \texttt{mx} and \texttt{ms} set the starting and maximum size of the memory allocation pool (the garbage collected heap). However, this did not work for the larger classes, nor for the 64 or 128 nodes runs that have been done to date.

4. Discussion/Interpretation

It should be noted that the use of the LAM MPI libraries has had an impact on the results, making analysis difficult. For every test run, the LAM versions of the tests were about 15\% slower than the C or Fortran version, which of course was reflected in the Java runtimes. The reason that LAM is slow is that it does not have access to the proprietary IBM high-speed communication protocol (only certain libraries have access to them), and is therefore confined to the slower TCP/IP protocol.

For the EP kernels, the JavaMPI programs scaled similarly to the Fortran version on 2 to 128 nodes. The measured performance of Java was within 50\% of the actual \texttt{f77-LAM} run times, and 40\% of the Fortran versions. Since the \texttt{f77-LAM} code ran about 15\% slower than the \texttt{f77-POE} version, this means that JavaMPI based on the IBM MPI library, the performance could increase by another 15\%, and approaching 50\% of the \texttt{f77-POE} codes.

Some of the reduced JavaMPI performance can be attributed to the overhead associated with using the LAM MPI library, so it is expected that Java performance would improve if the faster IBM MPI libraries could be used. The EP Kernel tests the floating-point calculation efficiency, so these results support the conclusions of the Java community, namely that the JVM needs improvements in the area of floating point arithmetic. However, some of these costs could also be attributed to the cost of object oriented programming: calls to the Math classes are external calls and are expensive. These costs may not be easily reduced.

As noted above, the Java IS data follow the C-POE curves out to about 32 nodes, but then a performance rollover begins to occur for both the C-LAM and JavaMPI data, for all three problem sizes. This effect has been observed before: similar roll-off effects due to calls to \texttt{MPI\_alltoallv} have been documented by the developers of the JavaMPI library on a similar system [15]. Mintchov reports that, when running LAM and JavaMPI on a 120 MHz IBM SP-2 system, they observed the same performance losses occurring at 16 nodes. After timing various parts of the IS codes used in these experiments, we found that most of the time spent is spent broadcasting the sorted keys within the call to \texttt{MPI\_alltoallv}, so our results are consistent their observations. Their solution was to create a modification to the LAM library to improve the \texttt{MPI\_alltoallv} efficiency. They report that this solution worked, but these patches were unavailable at the time this research was performed. Thus, we can expect that this roll-off effect could be eliminated or significantly reduced if the LAM patch were implemented, or we could eliminate the use of the LAM libraries completely.

As mentioned above, increases in the performance numbers for the IS Class A benchmark when achieved when jobs were run with the \texttt{java -mx1000m and -ms1000m} flags set as indicated. However, this did not work for the B or C classes for the 64-node case and none of the 128 node cases. Unfortunately, there was not enough machine time available to explore fully this issue, but there is definitely a problem with the size of the jobs as they run on a node. The size of the JVM program is about
5 times larger than the C or Fortran versions, when starting up on the node. The size of the job running on
the node grew continuously throughout the run-time, and for some classes this growth killed the job when
all the available memory was used up. This is could be due to an inefficient garbage collection algorithm.
This is a problem unique to Java, and improvements to the garbage collection algorithm will be needed in
order to increase the performance of this language.

5. Conclusions

The primary focus of this study was to implement and run well known HPC benchmarks in Java,
and then make a direct comparison of the performance of Java to Fortran or C implementations of the same
programs. The tests chosen for comparison were two of the NAS Parallel Kernel Benchmarks - the
Embarrassingly Parallel (EP) and the Integer Sort (IS) Kernels. All three problem sizes (A, B, and C) were
run for both of these benchmarks, on 2 to 128 nodes on the IBM SP-2, using standard MPI and JavaMPI
libraries.

The results of this study demonstrate that the performance of JavaMPI is within factors of 2-3 of
the Fortran or C equivalent, and that the performance scales well for nearly all cases. There is a drop-off
in performance for the Integer Sort, Class C problem size, which is observed for both the C-LAM and
JavaMPI codes. The drop-off is most likely associated with the performance of the LAM MPI libraries,
and partially Java or JavaMPI.

Based on the results of this research, we believe that there should be further investigations into the
use of Java both for high-performance and for scientific computing. Performance increases can be
expected when some of the well-known Java costs, such as those associated with array addressing and
garbage collection, are reduced. Additionally, commercially available compilers and JIT's should help
improve these performance numbers. Future research should include continued scalability studies (larger
systems), completion of the NAS JavaMPI benchmark suite (finish porting the NAS codes), and
investigation of other Java-based MPI libraries (mpiJava, MPIJ).

6. Acknowledgements

The authors would like to thank NPACI (The National Partnership for the Advanced
Computational Infrastructure) and The San Diego Supercomputer Center (SDSC) for the use of
their systems and for the technical support that they provided for this research.

7. References


1998.


