A Study of Java Networking Performance on a Linux Cluster

Mark Baker, Hong Ong, and Aamir Shafi
Distributed Systems Group
University of Portsmouth
Portsmouth, PO1 2EG, UK
{mark.baker, hong.ong, aamir.shafi}@port.ac.uk

Abstract

The Message Passing Interface (MPI) provides a common high-level Application Programming Interface (API) to implement parallel programs on commodity off-the-shelf (COTS) clusters. Recently, increasing attention has been shown in using Java-based MPI implementations. The reason for this growing interest is that the Java language offers a potentially machine-independent method of distributing the same code to perform computation on heterogeneous computer platforms. However, the communication between parallel Java programs is a critical issue for these types of application. Consequently, the Java communication performance issues must be addressed in order to provide an efficient Java based message-passing system. This paper focuses on the communication performance of Java and evaluates its at the lower-levels. We emphasize both methodology and evaluation guidelines in order to ensure reproducibility and consistent interpretations. Additionally, we also consider tuning the Java Virtual Machine (JVM) to improve the communication performance.

1. Introduction

Today, clusters [1][2] have become the standard platform for distributed and parallel computing in applied computer science. In particular, clusters provide a cost-effective alternative for applications requiring large quantities of computational power. Indeed, many application areas have replaced the traditional high performance computing (HPC) hardware with PC-based clusters. This is even more so for research groups and small to medium enterprise for which large scale HPC platforms are simply unaffordable.

Clusters, and other high-performance parallel computers, have traditionally been used for scientific applications, which predominantly use the C or Fortran language. However, there is a growing interest in using modern object-oriented languages, particularly Java, for implementing complex science and engineering applications. Java [3] has the potential to be a powerful language for developing such applications, partly because of its portability, its rich class libraries, and its support for distributed and concurrent computing paradigms. Although Java has been criticized for its performance limitations [4], this situation is changing rapidly with the advent of research and development into more efficient JVMs, use of multi-threading and other techniques to take advantage of modern computer architectures [5]. To date, Java has proven useful as a programming tool for cluster and distributed computing systems.

A common approach to parallel computing has been through the use of message passing. A notable achievement of the parallel computing community in 1990s was the emergence of a de facto standard interface for writing message-passing programs in procedural languages such as Fortran or C. This Message Passing Interface (MPI) [6][7] standard has now grown and subsumed many of the important ideas in message-based arena. More recently, a number of experimental research projects have built MPI related systems that support Java in some way. Many of these have attempted to provide Java bindings to underlying MPI [8][9][10] implementations. However, there are growing research efforts to provide a pure Java message-passing infrastructure [11][12].

Although today clusters are built using the same type of processors and communications technologies as those in traditional massively parallel systems, the message-passing performance between their nodes is much lower. Inefficiency occurs at many levels between the application and hardware layers. In the worst case, approximately 50% of the raw

bandwidth performance could be lost in the message-passing layer [13]. In addition, the operating system and device driver often add to the message latency and further decrease the maximum bandwidth by performing multiple memory-to-memory copies of data when messages are prepared for transmission. This extra data movement results in the contention of the memory bus, which typically occurs well before the Peripheral Component Interconnect (PCI) bus gets saturated. Therefore, the first step in improving the overall performance of a message passing system is to identify the potential performance bottlenecks [13] and possibly resolve them in the early development stage.

The Distributed Systems Group (DSG) is developing a pure Java message passing system based on the MPJ API specification [11]. Currently, we are focusing on the MPJ device level. Since this level mandates that non-blocking communication operations (i.e., `MPI_ISend` and `MPI_IRecv`) must be supported, we are investigating ways to provide an efficient "MPJ device" implementation. Among others, we are looking at the non-blocking features introduced in the New I/O APIs for the Java Platform [14]. However, prior to investing huge amount of development efforts, we would like to ascertain the benefit of using the Java New I/O package and compare its performance to the classic Java I/O. Consequently, the study reported in this paper concentrates on evaluating the lower-level communication performance, i.e., using the Java I/O (`java.io`) and the Java New I/O (`java.nio`) packages. The goal is to identify the potential bottleneck (if any) and provide a good generalized conclusion such that a more efficient MPJ runtime can be implemented.

This paper is organized as follow: Section 2 provides an overview of the benchmarking methodology used throughout the paper. The test environment used for our evaluation is described in Section 3. Subsequently, Section 4 presents and analyzes the performance results. Finally, Section 5 concludes this paper with a summary of our findings and makes a few remarks on future work.

2. Benchmarking Methodology

NetPIPE is a network protocol independent performance evaluation tool originally developed by Ames Laboratory [14]. It addresses issues such as the time required to transmit a given data block size to its destination, the maximum attainable throughput by applications, the maximum throughput for a given data block size, the communication overhead due to the legacy protocol stacks, and the latency of a communication protocol associated with a network interface. By addressing these issues, NetPIPE provides useful performance information for software developers to determine the best combination of a suitable communication protocol and an optimal buffer size for their communication bound applications. This can potentially help applications to achieve the maximum bandwidth and the minimum latency of the underlying network media.

NetPIPE uses ping-pong like transfer for each data block size. It increases the transfer block size from a single byte to large blocks until transmission time exceeds one second. Specifically, for each block size `c`, three measurements are taken for block sizes `c-p` bytes, `c` bytes and `c+p` bytes, where `p` is a perturbation parameter with a default value of `3`. This allows examination of block sizes that are possibly slightly smaller or larger than an internal network buffer.

The design of NetPIPE consists of the protocol independent driver, and a set of well-defined communication APIs. The communication APIs in turn implement the protocol specific module. Consequently, the protocol independent driver reuses the same APIs to evaluate and compare a variety of communication protocols. Currently, NetPIPE has been developed using the C and Java languages. The C version of NetPIPE has implemented the TCP, VIA [15][16], GM [17][18], PVM [19], and MPI communication protocol modules. The Java version of NetPIPE supports only the TCP protocol using the Java I/O package.

For the purpose of our performance evaluation, we have implemented the TCP protocol module using the Java New I/O packages. In Section 2.1, we describe the implementation and point out some desired features found in the Java New I/O package.

2.1. Java NIO NetPIPE Implementation

New I/O APIs for the Java Platform [14], defined in Java Specification Request (JSR) 51 [20], introduces new capabilities to the Java platform. Channels, defined in `java.nio.channels` package, is a new primitive I/O abstraction that

supports block-oriented I/O and can be configured to support multiplexed and non-blocking I/O. `java.nio` package defines `buffer`, which is a non-resizable container for a fixed amount of data of a specific primitive type. A `buffer` is either *direct* or *non-direct*. Given a direct buffer, the JVM will make a best effort to perform native I/O operations directly upon it.

We have implemented a Java version of NetPIPE based on the `java.nio` package. The implementation uses direct and indirect buffers to read/write blocks of data across the network. To make efficient use of the buffers, the Java NetPIPE pre-allocates a fixed chunk of memory that is reused each time for the read/write operations. For the send methods, the byte array to be sent is copied onto the `ByteBuffer` that is passed as an argument to `SocketChannel`'s write method. The implementation of the send data method is outlined in Table 1.

```
…
// copy bytes from the byte array
// onto ByteBuffer
buffer.put(bytes);

//flip the buffer
buffer.flip();

// write ByteBuffer into the
// channel
channel.write(buffer);
```

Table 1: The NetPIPE TCP `write` Method

The receive methods first reads `numberOfBytesRead` into the `ByteBuffer` and then copies them onto the internal byte `buffer` array. For a large message block, the receive method reads and appends the `numberOfBytesRead` read in each loop into the byte `buffer` array until the receiver has read the block of data it is expecting to receive. The implementation of the receive data method is shown in the code in Table 2.

```
do {
   …
//read bytes from channel into ByteBuffer
   int numberOfBytesRead =                    channel.read(buffer);
   …
//copy and append bytes onto byte array
buffer.get(bytes,start,numberOfBytesRead;
…
}  while (more bytes to read);
```

Table 2: The NetPIPE TCP `read` method

3. Test Environment

3.1. Hardware/Software

The test environment for collecting the performance results was conducted on a cluster of 8 dual Intel Xeon 2.8 GHz PCs using the Intel E7501 chipset. The PCs were equipped with 2 Gbytes ECC RAM with 533 MHz Front Side Bus (FSB).

The motherboard (SuperMicro X5DPR-iG2) is equipped with 2 onboard Intel Gigabit LAN adaptors with one 64-bit 133 MHz PCI-X slot and one 64-bit 66 MHz PCI slot. The PCs were connected together through a 24-port Ethernet switch. In addition, two PCs were connected back-to-back via the onboard Intel Gigabit adaptors.

The PCs were running the Red Hat 9.0 distribution with the 2.4.20 Linux kernel. The software used for the Intel Gigabit adaptor was the proprietary Intel e-1000 device driver. For compiling the Java version of NetPIPE (both Java I/O and New I/O implementation), the Java SDK 1.4.2 was used; GNU GCC 3.2 was used to compile the C version of NetPIPE.

3.2. TCP Tuning

Since TCP was originally engineered to provide a general transport protocol, it is not by default optimised for streams of data coming in and out of the system at high transmission rates (e.g., 1Gbit/sec). Farrell et. al. [21] have shown that communication performance is effected by a number of factors and indicated that one can tune certain network parameters to achieve higher TCP/IP performance, especially for a high speed network such as a Gigabit Ethernet network. We have taken care to tune the TCP parameters according to RFC 1323 TCP/IP Extension for High Performance [22].

4. Performance Results

Benchmarking results (e.g., Java memory and threads performance) have been collected to evaluate the various performance aspects of the Java runtime. However, for the purpose of this paper, we will only present the TCP throughput and latency performance for the Fast and Gigabit Ethernet network. For the rest of this section, we will denote the three NetPIPE implementations as C-NP as the C version, JIO-NP as the Java version using I/O package, and JNIO-NP as the Java version using New I/O package. Additionally, unless stated otherwise, the JNIO-NP uses direct buffer allocation for storing messages.

For timing measurement, C-NP uses the system `gettimeofday()`, JIO-NP and JNIO-NP use the `System.currentTimeMillis()` from the `java.lang` package. Although the millisecond resolution is less than ideal for measuring performance, the NetPIPE runtime for each data block is sufficiently long that the clock resolution is not significant. Additionally, NetPIPE also reports the standard time variance for each data block, and thus provide a good indication if the timing measurement is unreliable. However, for all tests, we observed that the time variance is either zero or close to zero.

4.1. Throughput Performance

Figure 1 shows the TCP bandwidth performance on Fast Ethernet network using a socket buffer size equal to 16Kbytes (Figure 1a) and 256Kbytes (Figure 1b). It should be noted that the default socket buffer size for Linux kernel 2.4 is 16Kbytes.

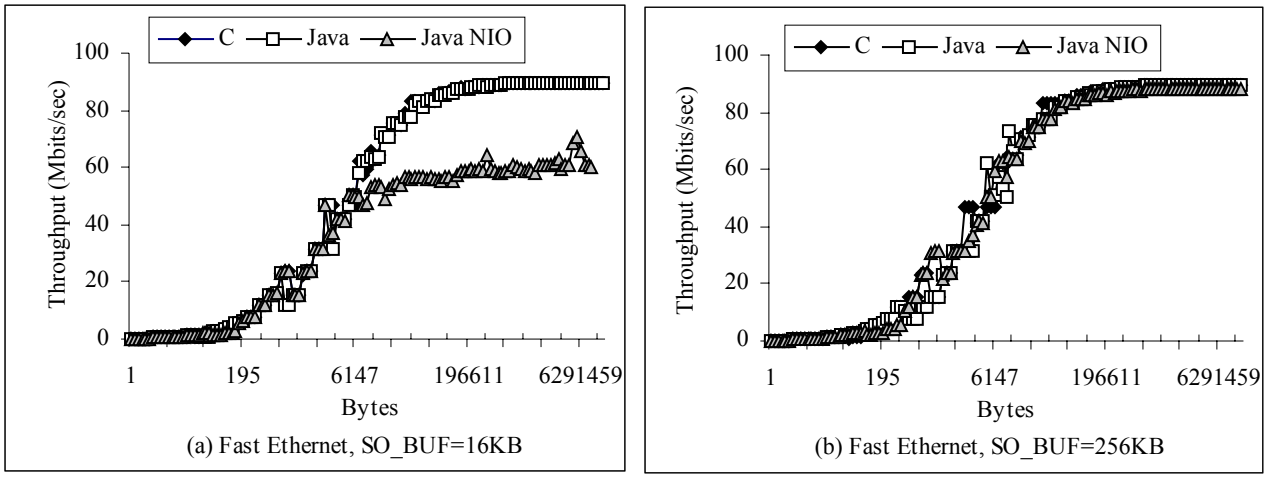

Figure 1 TCP Throughput Performance on Fast Ethernet Network

Using socket buffer size equal to 16Kbytes, the C-NP (labeled as C) achieves a maximum TCP throughput of approximately 89 Mbits/sec. The JIO-NP (labeled as Java) performs equally well as the C-NP. On the other hand, the JNIO-NP (labeled as Java NIO) only attains a maximum throughput of approximately 60 Mbits/sec. There are two interesting observations here. First, the JIO-NP matched the C-NP TCP performance. Second, for data block size less than 8KB, the JNIO-NP behaves similar to JIO-NP. However, for data block size greater than 8KB, JNIO-NP does not perform as well as JIO-NP. This suggested that JNIO-NP spent a considerable amount of time in traversing the JVM layer before large data blocks are sent.

The maximum amount of data ("bandwidth x delay") that can be transmitted over the Fast Ethernet link (for our cluster) is roughly 109 Kbytes. Thus, increasing the default socket to 256Kbytes will not improve the TCP performance. As expected, the C-NP and JIO-NP TCP performance remain the same when the socket buffer size is set to 256Kbytes (see Figure 1b). However, for JNIO-NP, the maximum throughput has increased to 89 Mbits/sec. An obvious explanation for this is that by increasing the socket buffer size the overhead incurred in copying the message buffer from a Java application (JNIO-NP) to the system buffer has been overlapped. Thus, hiding the actual transmission cost from the Java application. Regardless of socket buffer size, the latency of C-NP, JIO-NP, and JNIO-NP for the Fast Ethernet is 125 microseconds.

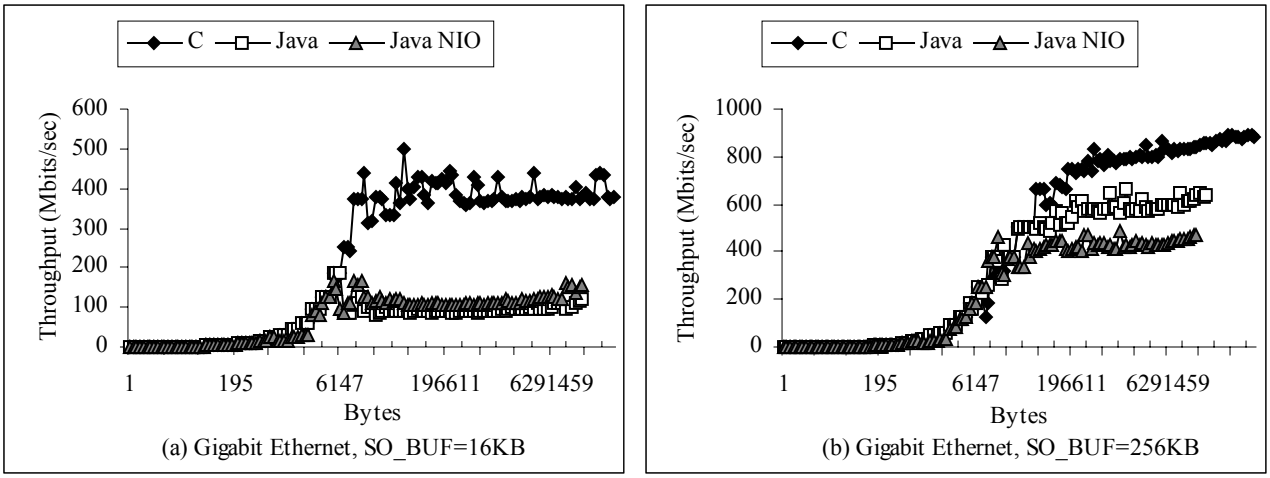

Figure 2: TCP Throughput Performance on Gigabit Ethernet Network

Figure 2 shows the TCP throughput performance on a Gigabit Ethernet network. Using socket buffer size equals to 16Kbytes (Figure 2a), the maximum attainable TCP bandwidth is approximately equal to 436Mbits/sec for C-NP, and 112Mbits/sec for both JIO-NP and JNIO-NP. The latency for all three is approximately 125 microseconds. Increasing the socket buffer size to 256Kbytes improves all the NetPIPE implementations. In particular, the C-NP attains a maximum throughput of 889Mbits/sec, whereas the JIO-NP and JNIO-NP attain approximately 644Mbits/sec and 477Mbits/sec, respectively. Note that the JNIO-NP performance is slightly better than JIO-NP for socket buffer size equals to 16 Kbytes. When the socket buffer size increases, the JIO-NP TCP performance surpassed the JNIO-NP. On the other hand, JIO-NP and JNIO-NP are about 76% and 54% of the C-NP TCP performance, respectively. For the Gigabit network with socket buffer size equals to 256Kbytes, the latency for all three is approximately 63 microseconds.

Again, like the Fast Ethernet, a possibility to improve the Java networking performance and to fully utilize the potential gigabit bandwidth is to further increase the socket buffer size. However, to achieve the same effect as the Fast Ethernet, the required socket buffer size would have to be set to roughly 18Mbytes for the Gigabit network. This is obviously a waste of resources. Consequently, other performance tuning techniques must be considered.

4.2. Tuning the JVM Heap Memory Size and the Garbage Collection Method

Sun's JVM offers a variety of command line switches that can be used to tune the memory allocation and garbage collection behavior [23]. In an attempt to improve Java networking performance, we performed a series of experiments ranging from tuning the JVM heap memory sizes to using different garbage collection methods. In general, most of these

efforts did not help in improving the networking performance. Hence, we will only highlight the results collected from memory allocation and garbage collection tuning.

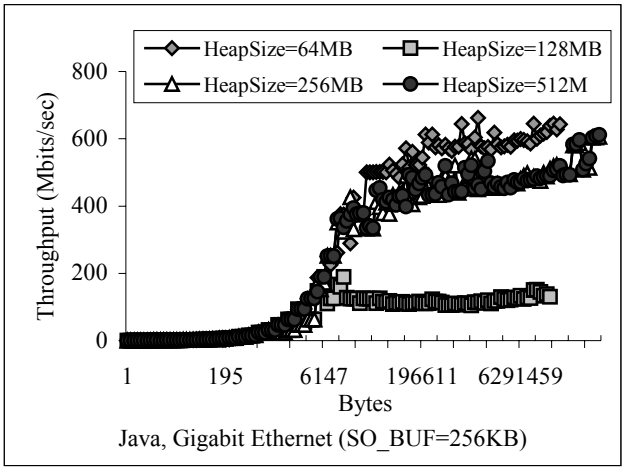

Figure 3: Comparing Java Networking Performance Using Different Heap Memory Size

The memory allocation tuning experiment we conducted was to set the JVM heap memory to various sizes. Figure 3 shows the performance results collected on a Gigabit Ethernet with socket buffer size set to 256 Kbytes. From the figure, it is obvious that setting different heap memory sizes do not increase the JIO-NP network performance to the level of C-NP. In fact, with heap memory size equals to 128 Mbytes, the JIO-NP performance actually drops to less than 120 Mbits/sec.

The Java SDK 1.4.2 comes with three garbage collectors in addition to the default collector. The three garbage collectors are the throughput, concurrent, and incremental collectors. Each of the collectors is intended for different application's performance requirements. As the name implied, the throughput collector improves the throughput of the application by allowing the collector to run in a separate thread than the application. The concurrent collector reduces the amount of time the application has to wait (known as paused time) when the garbage collection is in progress. Finally, the incremental collector is similar to the concurrent collector except that it spreads the large paused time of a major collection over many minor collections. We have tested the three collectors and found no improvement. Figure 4 shows the performance results using the incremental garbage collector.

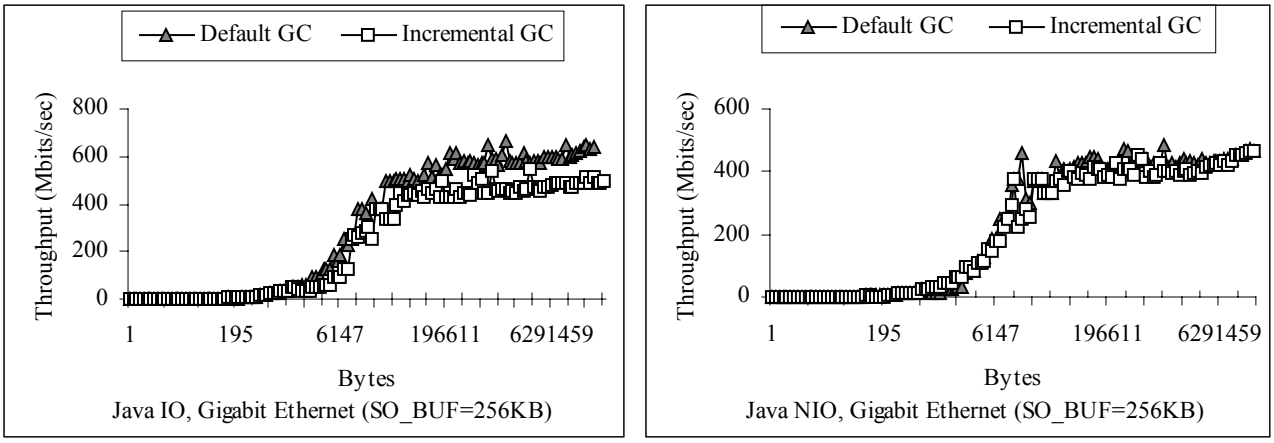

Figure 4: Comparing Java Network Performance using Default GC to Incremental GC

4.3. Java NIO Direct vs. Indirect Buffer Issue

A `ByteBuffer` can be direct or indirect. A direct buffer is allocated from the native systems memory and thus is not subject to garbage collection like normal Java objects. As a result, a direct buffer has a higher creation cost than indirect buffer creation, but it is supposed to have faster I/O operations than the indirect buffers. Figure 5 compares the performance of JNIO-NP using the direct buffer to indirect buffer. The results indicated that there is no difference in using either direct or indirect buffers.

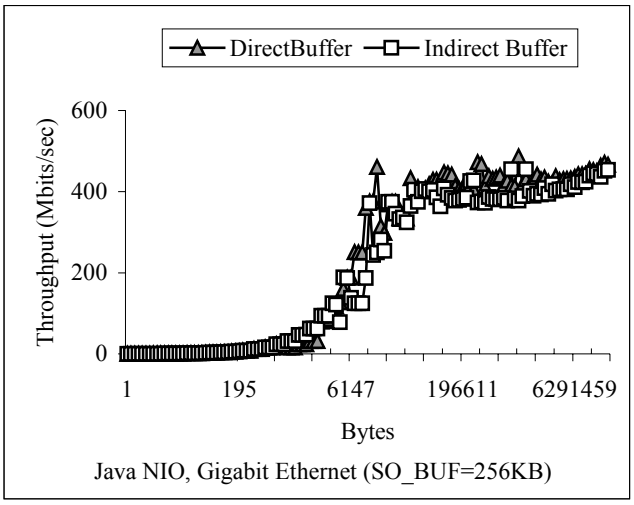

Figure 5: Java NIO Direct Buffer vs. Indirect Buffer

5. Conclusions

In this paper, we have investigated the Java networking performance using the NetPIPE, implementing communications with the classic Java I/O and the New I/O packages. We have compared Java networking performance against the C NetPIPE implementation on a Fast Ethernet and Gigabit Ethernet network. The performance results show that on the Fast Ethernet network the JIO-NP TCP performance is comparable with the C-NP. However, the JNIO-NP TCP performance is slower than both JIO-NP and C-NP. The reason being that on a slower network the overhead of traversing the JVM layers is not obvious to the Java applications. However, on a faster network, i.e., the Gigabit network, these overheads were exposed. In particular, both the JIO-NP and JNIO-NP perform poorly as compared to their C counterpart.

A lot effort was expended tuning the JVM in the hope of improving the Java networking performance. However, the tuning had very little effect on improving performance. In particular, changing the heap memory size and using different garbage collectors did not improve the network performance of Java. In addition, we have tested both direct and indirect buffers available in the Java New I/O package. There is no indication that using an indirect buffer can improve the Java TCP throughput performance. However, one has to bear in mind that the NetPIPE implementation is a single threaded, point-to-point communication application. In that sense, it has not utilized other desirable features of New I/O package such as asynchronous send/receive and gather/scatter type of operations. We expect that in a multi-threaded application, Java New I/O could potentially perform better than the Java I/O. However, the Java communication performance still remains an issue. In order to build an efficient MPJ message-passing system, further work needs to address this issue, perhaps by exploring different communication techniques.

We are aware that the NetPIPE TCP performance results reported in this paper contribute a single view of the overall Java performance, i.e., how the Java communication performs. An immediate task for this project is to combine our other Java benchmarking results to form a global view of Java performance for distributed computing. Indeed, a multi-threaded MPJ runtime system using Java New I/O is under development.

References

[1] R. Buyya, editor, "High Performance Cluster Computing, volume 1: Architectures and Systems", *Prentice-Hall*, ISBN 0-13-13784-7, 1999.

[2] R. Buyya, editor, "High Performance Cluster Computing, volume 2: Programming and Applications", *Prentice-Hall*, ISBN 0-13-13785-5, 1999.

[3] J. Gosling, B. Joy, and G. Steele, "The Java Language Specification. The Java Series", *Addison Wesley Longman*, ISBN 0-201-63451-1, 1996.

[4] J. A. Mathew, P. D. Coddington, and K. A. Hawick, "Analysis and Development of Java Grande Benchmarks", *In Proceeding of the ACM 1999 Java Grande Conference*, April 1999.

[5] Java Grande Forum. "Java Grande Forum Report: Making Java work for high-end computing", *Technical Report TR-01, Java Grande Forum*, November 1998. http://www.javagrande.org.

[6] The MPI Standard., http://www.mpi-forum.org/

[7] MPI Forum, "MPI: A message-passing interface standard", *International Journal of Supercomputer Applications*, 8(3/4), pp 165-416, 1994

[8] M. A. Baker, B. Carpenter, G. Fox, S-H. Ko, and X. Li, "mpiJava: A Java MPI interface", *In Proceeding of the International Workshop on Java for Parallel and Distributed Computing* (IPPS/SPDP), February 1999.

[9] K. Dincer, "Ubiquitous message passing interface implementation in Java: JMPI", *In Proceeding 13th International Parallel Processing Symposium and 10th Symposium on Parallel and Distributed Processing*. *Institute of Electrical and Electronics Engineers*, 1998.

[10] G. Judd, M. Clement, and Q. Snell, "DOGMA: Distributed Object Group Metacomputing Architecture", *Concurrency: Practice. And Experience*, 10(11-13),(1998)

[11] M. A. Baker and B. Carpenter, "MPJ: A Proposed Java Message-Passing API and Environment for High Performance Computing", *the 2nd Java Workshop at IPDPS 2000*,Cancun, Mexico, *LNCS*, Springer Verlag, Germany, pp 552 - 559, May 5th 2000, ISBN 3-540-67442-X

[12] M. A. Baker, D.B Carpenter, et al., mpiJava: An Object-Oriented Java interface to MPI, the *1st Java Workshop at the 13th IPPS & 10th SPDP Conference*, Puerto Rico, April 1999, *LNCS*, Springer Verlag, Heidelberg, Germany, ISBN 3-540-65831-9.

[13] H. Ong and P. A. Farrell, "Performance Comparison of LAM/MPI, MPICH, and MVICH on a Linux Cluster connected by a Gigabit Ethernet Network", *In Proceeding of the 4th Annual Linux Showcase & Conference, Atlanta*, October 10-14, 2000

[14] Sun MicroSystems, "New I/O APIs for the JavaTM Platform Specifications", *Sun MicroSystems*, 2002. http://java.sun.com/j2se/1.4.2/docs/guide/nio/index.html.Quinn O. Snell, Armin R. Mikler, and John L.Gustafson, "NetPIPE: Network Protocol Independent Performance Evaluator", *IASTED Conference*, Ames Lab., Scalable Computing Lab., Iowa State, 1997. http://www.scl.ameslab.gov/netpipe

[15] Compaq Corporation, Intel Corporation, and Microsoft Corporation, "Virtual Interface Architecture Specification, Version 1.0", 1997.

[16] M. A. Baker, P. A. Farrell, H. Ong, and S. L. Scott, "VIA Communication Performance on a Gigabit Ethernet Cluster", *In Proceedings of the 7th International Euro-Par Conference*, Manchester, UK. Lecture Notes in Computer Science, vol. 2150/2001, Springer Verlag.

[17] N. J. Boden, D. Cohen, R. E. Felderman, A. E. Kulawik, C. L. Seitz, J. N. Seizovic, and W-K. Su. Myrinet: A Gigabit-per-Second Local Area Network. *IEEE MICRO*, Vol. 15 No. 1, pp 29-36, 1995.

[18] Myricom, "The GM-2 Message Passing System", http://www.myri.com

[19] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam, "PVM: Parallel Virtual Machine, *MIT press*, 1994. http://www.csm.ornl.gov/pvm

[20] Sun MicroSystems, "JSR 51 - New I/O APIs for the Java™ Platform", *Sun MicroSystems*, 2002. http://www.jcp.org/jsr/detail/51.jsp.

[21] P. A. Farrell and H. Ong, "Communication Performance over a Gigabit Ethernet Network", *IEEE 19th Proceeding of IPCCC*, 2000.

[22] Jacobson, Braden and Borman, "TCP Extensions for High Performance", *RFC 1323*, 1992.

[23] Sun MicroSystems, "Tuning Garbage Collection with the 1.4.2 Java Virtual Machine", 2003. http://java.sun.com/docs/hotspot/gc1.4.2