An Approach to Buffer Management in Java HPC Messaging

Mark Baker, Bryan Carpenter[1], Aamir Shafi
Distributed Systems Group, University of Portsmouth

Abstract

One of the most challenging aspects to designing a Java messaging system for HPC is the intermediate buffering layer. The low-level communication device and higher levels of the messaging software use this buffering layer to write and read messages. The Java New I/O package adds the concept of direct buffers, which—coupled with a memory management algorithm—opens the possibility of efficiently implementing this buffering layer. In this paper, we present our buffering strategy, which is developed to support efficient communications and derived datatypes in MPJ Express – our implementation of the Java MPI bindings. Our memory management system uses Knuth's buddy algorithm for optimal memory allocation. We evaluate the performance of our buffering layer and demonstrate the usefulness of direct byte buffers in a Java messaging system.

1. Introduction

A challenging aspect of implementing Java HPC messaging software is providing an efficient intermediate buffering layer. The low-level communication device and higher levels of the messaging software use this buffering layer to write and read messages. The heterogeneity of these low-level communication devices poses additional design challenges. To appreciate this fully, assume that the user of a messaging library sends ten elements of an integer array. The C programming language can retrieve the memory address of this array and pass it to the underlying communication device. If the communication device is based on TCP, it can then pass this address to the sockets write method. For proprietary networks, like Myrinet [4], this memory region can be registered for Direct Memory Access (DMA) transfers, or copied to a DMA capable part of memory and sent using low level Myrinet communication methods.

On the other hand, if a user of a Java messaging system sends an array of ten integers, these have to be copied to a `ByteBuffer`, which is used as an argument to the `SocketChannel` write method. Here we are only considering the Java New I/O (NIO) [14] package as we are convinced that NIO provides essential ingredients [1] of an efficient messaging system via non-blocking I/O and direct buffers. For proprietary networks like Myrinet and Infiniband, NIO provides a viable option because it is now possible to get memory addresses of

[1] Open Middleware Infrastructure Institute (OMII), University of Southampton

direct `ByteBuffers`, which can be used to register memory regions for DMA transfers. Using direct buffers may eliminate the overhead [2] incurred by additional copying when using JNI with JVMs that do not support pinning. On the other hand, it may be preferable to create a native buffer using the Java Native Interface (JNI) [13]. These native buffers can be useful for a native MPI or a proprietary network device.

Based on these factors, we have designed an extensible buffering layer that allows various implementations based on different storage mediums like direct or indirect `ByteBuffer`s, byte arrays, or memory allocated in the native C code. The higher levels of the MPJ Express (MPJE) [5] use the buffering layer through an interface. This implies that functionality is not tightly coupled to the storage medium. The motivation behind developing different implementations of buffers is to achieve optimal performance for lower level communication devices.

The buffering layer developed provides variants of write and read methods. It also supports gather and scatter functionality that provides the basis of support for MPI-like derived datatypes. Implementing these features in a Java messaging system is fairly unique because derived datatypes were introduced in the MPI standard for languages like C/Fortran. The derived datatypes can be used for efficient sending and receiving of non-contiguous sections of user data. Also, using derived datatypes helps avoid the overheads of Java object serialization and de-serialization. We refer interested readers to Appendix 1 for an overview of MPJE.

A closely related buffering API with the same gather and scatter functionality was originally introduced for Java in the context of Adlib communications library used by HPJava [7]. In the current work, we have extended this API to support the derived datatypes in a fully functional MPI interface.

Our buffering strategy uses a pooling mechanism for avoiding creating an instance of buffer for each communication method. The creation time of these buffers can affect overall communication time, especially for large messages. Our current implementation is based on Knuth's buddy algorithm [12], but it is possible to use other pooling techniques.

The main contribution of this paper is the design and implementation of our buffering layer for HPC supported by two different pooling mechanisms. We have evaluated the performance of these two pooling mechanisms. We show that one of them is faster with a smaller memory footprint. Also, we demonstrate the usefulness of direct byte buffers in Java messaging systems.

This paper is organized as follows. Section 2 discusses related work. The strategy itself with an explanation of our memory management algorithms is described in section 3. In section 4, we present performance evaluation of our buffering strategies. Section 5 concludes the paper outlining future research work.

2. Related Work

The most popular Java messaging system is mpiJava [6], which uses a JNI wrapper to the underlying native C MPI library. Being a wrapper library, mpiJava does not use a clearly distinguished buffering layer. After packing a message onto a contiguous buffer, a reference to this buffer is passed to the native C library. But in achieving this additional copying is required between the JVM and the native C library if the JVM does not support pinning of memory. This overhead is especially noticeable for large messages.

Javia [3] is a Java interface to the Virtual Interface Architecture (VIA). An implementation of Javia exposes communication buffers used by the VI architecture to Java applications. These communication buffers are created outside the Java heap and can be registered for DMA transfers. This buffering technique makes it possible to achieve performance within 1% of the raw hardware.

An effort similar to Javia is JAGUAR [2]. This uses compiled-code transformations to map certain Java bytecodes to short, in-lined machine code segments. These two projects, JAGUAR and Javia were the motivating factors to introduce the concept of direct buffers in the NIO package. The design of our buffering layer is based on direct byte buffers. In essence, we are applying the experiences gained by JAGUAR and Javia to design a generic and efficient buffering layer that can be used for pure Java and proprietary devices in Java messaging systems alike.

3. The Buffering Layer in MPJE

In this section, we discuss our approach to designing and implementing an efficient buffering layer supported by a pooling mechanism. The self-contained API developed as a result is called the MPJ Buffering (`mpjbuf`) API. The functionality provided includes packing and unpacking of user data. The primary difficulty in implementing this is that the sockets do not directly access the memory and thus are unable to write or read the basic datatypes. The absence of pointers and the type safety features of the Java language make the implementation more complex. Most of the complex operations used at the higher levels of the library, such as communicating objects and handling gather or scatter operations, are also supported by this buffering layer.

3.1 The Layout of Buffers

An `mpjbuf` buffer object contains two data storage structures. The first is a static buffer, in which the underlying storage primitive is an implementation of the `RawBuffer` interface. The implementation of static buffer called `NIOBuffer` uses direct or indirect `ByteBuffer`s. The second is a dynamic buffer where a byte array is the storage primitive. The size of the static buffer is predefined, and can contain only primitive datatypes. The rationale behind this is that it is possible to calculate the number of bytes required before copying the data onto the buffer. The dynamic buffer is used to store serialized Java objects, where it is not possible to determine the length of the serialized objects beforehand. As a convenience to the user, Java basic datatypes may also be stored in dynamic buffers. In general, writing to or reading from a static section of the buffer is much faster than the corresponding operation on a dynamic section of the buffer.

A message consists of zero or more sections. Each section can hold elements of the same type, basic datatypes or Java objects. A section consists of a header, followed by the actual data payload. The data stored in a static buffer can be represented as big-endian or little-endian. This is determined by the encoding property of the buffer, which takes on of the values `java.nio.ByteOrder.BIG_ENDIAN` or `java.nio.ByteOrder.LITTLE_ENDIAN`. Figure 1 shows the overall layout of static buffer. The alignment provided by the padding at the end of each section supports low level APIs that require it for efficiency.

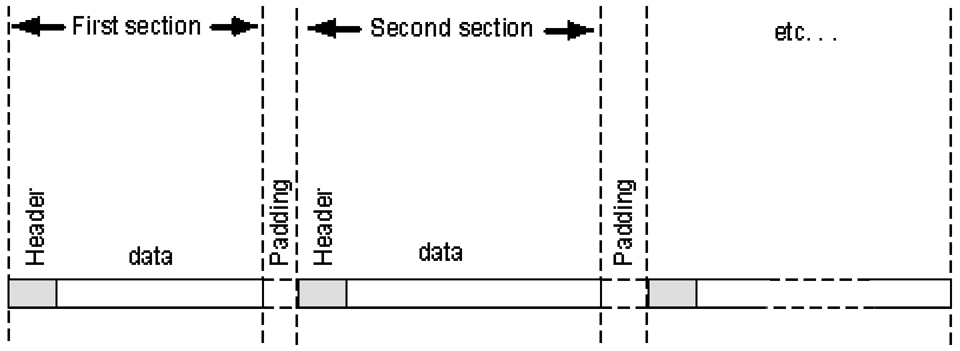


Figure 1: The Layout of a Static Buffer

As shown in Figure 1, a message consists of zero or more sections. The message consists of a message header followed by the data payload. Padding of up to 7 bytes may follow a section if the total length of the section (header + data) is not a multiple of `ALIGNMENT_UNIT`, which has value 8. The general layout of an individual section in the static buffer is shown in Figure 2.

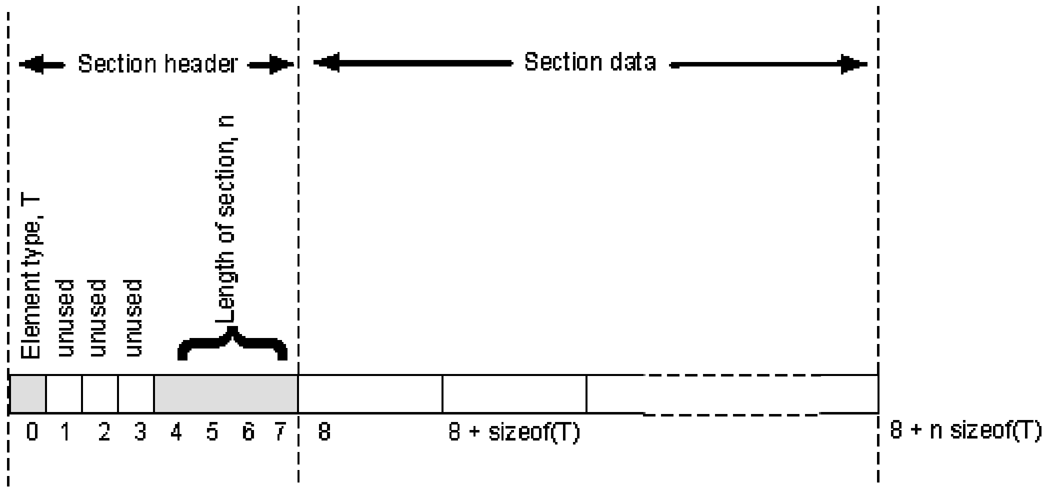


Figure 2: The Layout of a Single Section

3.2 The API

The most important class of the package used for packing and unpacking data is `mpjbuf.Buffer.` This class provides two storage options. For static sections it is possible to have alternative implementations. Figure 3 shows two implementations of `mpjbuf.RawBuffer` interface. The first, `mpjbuf.NIOBuffer` is an implementation based on `ByteBuffer`s. The second, `mpjbuf.NativeBuffer` is an implementation for native MPI device, which allocates memory in the native C code. Figure 3 shows the primary buffering classes in `mpjbuf` API.

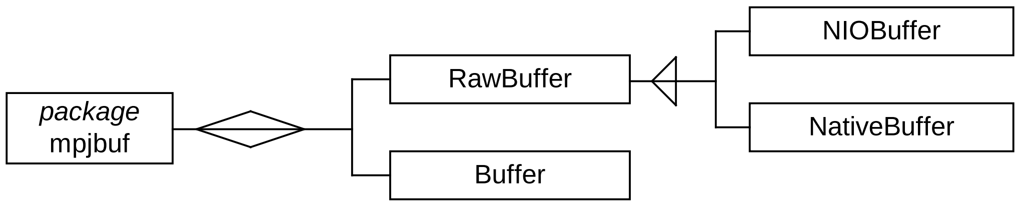


Figure 3: Primary Buffering Classes in `mpjbuf`

There are three basic kinds of method for writing data to a buffer section and for reading data from a buffer section. These methods are summarized in Table 1.

Packing/Unpacking Methods	**Section writing methods**	**Section reading methods**
1. Write/Read Methods	`write(type[] src, int srcOff, int numEls)`	`read(type[] dest, int dstOff, int numEls)`
2. Gather/Scatter Methods	`gather(type[] src, int numEls, int idxOff, int[] indexes)`	`scatter(type[] dest, int numEls, int idxOff, int[] indexes)`
3. Gather and Scatter Methods for multi-strided regions	`strGather(type[] src, int srcOff, int rank, int exts, int srs, int[] shape)`	`strScatter(type[] dest, int dstOff, int rank, int exts, int srs, int[] shape)`

Table 1: Buffer Packing or Unpacking Methods

The methods shown in row 1 of Table 1 are used to write and read contiguous Java arrays of all the primitive datatypes including object arrays. The methods shown in row 2 of Table 1 are used to write and read non-contiguous Java arrays of all the primitive datatypes including object arrays. The methods shown in row 3 of Table 1 likewise transfer data from or to a subset of elements of a Java array, but in these cases the selected sub-set is a "multi-strided region" of the Java array. The specification is fairly complex, but these are useful operations for dealing with multi-dimensional data structures, which occur often in scientific programming.

A buffer object has two modes – write and read. The write mode allows the user to copy the data onto the buffer, and the read mode allows the user to read the data from the buffer. It is not permitted to read from the buffer when it is in writeable mode. Similarly, it is not permitted to write to a buffer when it is readable mode.

3.3 Memory Management

We have implemented our own application level memory management mechanism based on a buddy allocation scheme [12]. The motivation is to avoid creating an instance of a buffer (`mpjbuf.Buffer`) for every communication operations like `Send()` or `Recv()`, which may dominate the total communication cost, especially for large messages. We can make efficient use of resources by pooling buffers for future reuse instead of letting the garbage collector reclaim the buffers and create them all over again.

The functionality provided by the buffering API is exported to the users through a `BufferFactory`. Table 2 shows the functionality provided by this buffer factory. Note that this factory returns `mpjbuf.RawBuffer`, which is used for storing static sections of the messages. Because the size of dynamic sections cannot be calculated beforehand, it is not possible to use pooling for dynamic sections of a buffer.

```
public static void init(String poolingAlgo)
public static mpjbuf.RawBuffer create(int size)
public static void destroy(mpjbuf.RawBuffer buffer)
public static void shut()
```

Table 2: Methods of `mpjbuf.BufferFactory`

3.3.1 General Explanation of Buddy Algorithm

In this section, we will briefly review Knuth's buddy algorithm. In our implementation, available memory is divided into a series of "regions". Each region has a storage medium associated with it—direct or indirect `ByteBuffer`. Initially, there is no region associated with the `BufferFactory`. Whenever a buffer is requested, the factory checks whether there is a region with size greater than the requested size. If a region does not exist or does not have free space, a new region is created. For managing the buffers, there is a doubly linked list `FreeList.` This `FreeList` refers to buffers at all possible levels starting from 0 to log2N (`region_size`). The level of a buffer can be thought of an integer, which increases as log2N (buffer_size) increases. We will discuss regions along with `FreeList`s and level in more detail in Section 3.3.2.

After finding or creating an appropriate region that can serve this request, the algorithm attempts to find a free buffer at the requested or higher level. If the buffer found is at a higher level, it is divided into two buddies and this process is repeated until we reach the required level. The `BufferFactory` returns the first free buffer at this level. Every allocated buffer is aware of its offset and the size of the region it belongs to. Figure 4 shows the allocation events of 1 Megabytes of block when the initial region size is 8 Megabytes.

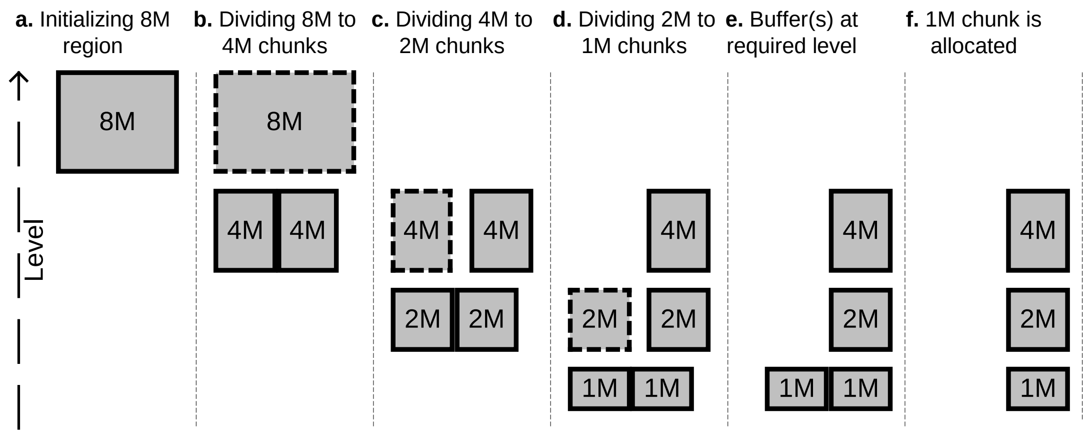

a. Initializing 8M region
b. Dividing 8M to 4M chunks
c. Dividing 4M to 2M chunks
d. Dividing 2M to 1M chunks
e. Buffer(s) at required level
f. 1M chunk is allocated
Level
8M
8M
4M
4M
4M
4M
4M
4M
4M
2M
2M
2M
2M
2M
2M
1M
1M
1M
1M
1M

Figure 4: Allocating a 1M Buffer

When the buffer is de-allocated, an attempt is made to find the buddy of this buffer. If the buddy is free, the two chunks of memory are merged together to form a buffer at the higher level. Once we have a buffer at the higher level, we execute the same process recursively until we do not find a buddy for the buffer at the higher level. Figure 5 shows the de-allocation events when a block of 1 Megabytes is returned to the buffer factory.

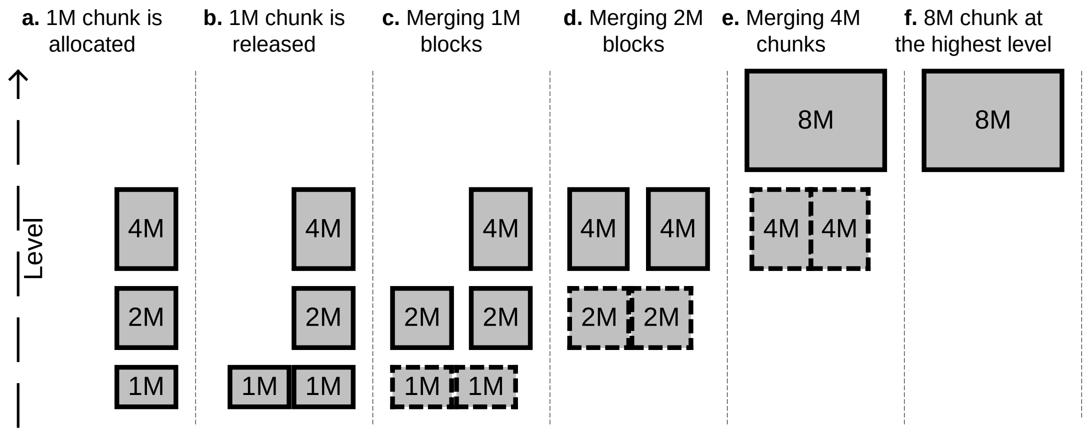


Figure 5: De-allocating a 1M Buffer

3.3.2 Two Implementations of Buddy Allocation Scheme for `mpjbuf`

In the MPJ buffering API it is possible to plug in different implementations of buffer pooling. A particular strategy can be specified during the initialisation of `mpjbuf.BufferFactory`. Each implementation can use different data structures like trees or doubly linked lists. In the current implementation, the primary storage buffer for `mpjbuf` is an instance of `mpjbuf.NIOBuffer`. Each `mpjbuf.NIOBuffer` has an instance of `ByteBuffer` associated with it. The pooling strategy boils down to reusing `ByteBuffer`s encapsulated in `mpjbuf.NIOBuffer`.

Our implementation strategies are able to create smaller thread-safe `ByteBuffer`s from the initial `ByteBuffer` associated with the region. We achieve this by using `BytBuffer.slice()` for creating new byte buffer whose contents are a shared sub sequence of original buffer's contents. In the sub-sections to follow, we discuss two implementations of memory management techniques.

The First Pooling Strategy

Our first implementation (hereafter called `Buddy1`) is developed with the aim of keeping a small memory footprint of the application. This is possible because a buffer only needs to know its offset in order to find its buddy. This offset can be stored at the start of the allocated memory chunk. If a user requests `s` bytes, the first strategy allocates `s+buddy_overhead` bytes buffer. The additional `buddy_overhead` bytes will be used to store the buffer offset. Also, the data structures do not store buffer abstractions like `mpjbuf.NIOBuffer` in the linked lists.

Figure 6 outlines the implementation details of our first pooling strategy. `FreeLists` is a list referring to `BufferLists`, which contains buffers at different levels. Here, level refers to the different sizes of buffer available. If a buffer is of size `s`, then its corresponding level will be log2N (`s`). Initially, there is no region associated with `FreeLists`. An initial chunk of memory of size `region_size` is allocated. At this point, `BufferLists` are created starting from 0 to log2N (`region_size`). When buddies are merged, a buffer is added to the `BufferList` at the higher level and the buffer itself and its buddy are removed from the `BufferList` at the lower level. Conversely, when a buffer is divided to form a pair of buddies, a newly created buffer and its buddy is added to the `BufferList` at the lower level while removing a buffer that is divided from the higher level `BufferList`.

An interesting aspect of this implementation is that `FreeList` and `BufferLists` are independent of a region and these lists grow as new regions are created to match user requests.

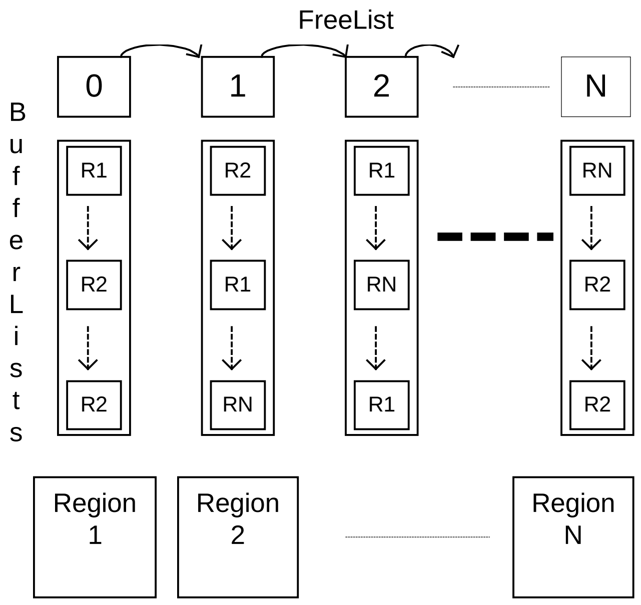


Figure 6: The First Implementation of Buffer Pooling

The Second Pooling Strategy

Our second implementation (hereafter called `Buddy2`) stores higher-level buffer abstractions (`mpjbuf.NIOBuffer`) in `BufferLists`. Unlike the first strategy, each region has its own `FreeList` and has a pointer to the next region as shown in Figure 7. While finding an appropriate buffer for a user, this implementation starts sequentially starting from the first region until it finds the requested buffer or creates a new region. We expect some overhead associated with this sequential search. Another downside for this implementation is a bigger memory footprint.

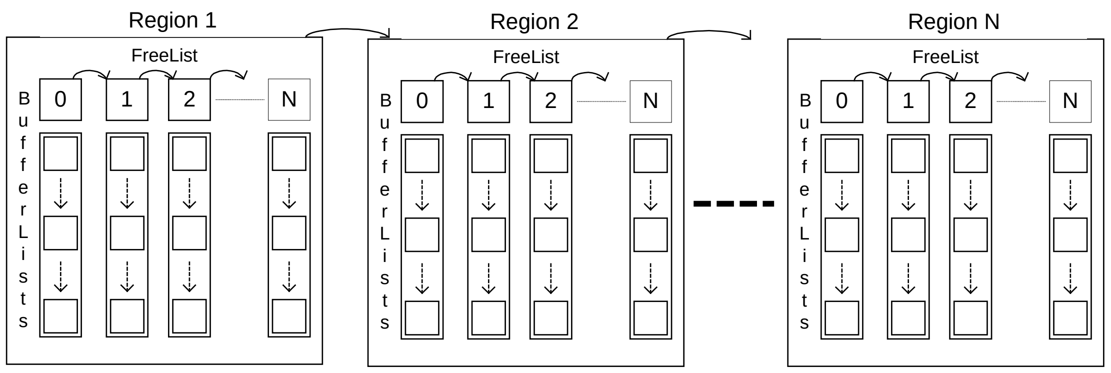


Figure 7: The Second Implementation of Buffer Pooling

4. Buffering Layers Performance Evaluation

In this section, we compare the performance of our two buffering strategies with direct allocation of `ByteBuffer`s. Also, we are interested in exploring the performance difference between using direct byte buffers and indirect byte buffers in MPJE communication methods. There are six combinations of our buffering strategies that will be compared in our first test – `Buddy1` using direct byte buffers, `Buddy1` using indirect byte buffers, `Buddy2` using direct byte buffers, `Buddy2` using indirect byte buffers, simple-minded allocation of NIO direct byte buffers, and simple allocation of indirect byte buffers.

4.1 Simple Allocation Time Comparison

In our first test, we are interested in comparing isolated allocation times for a buffer for our six allocation approaches. Only one buffer is allocated at one time throughout the tests. This means that after measuring allocation time for a buffer, it is de-allocated in the case of our buddy schemes (forcing buddies to merge into original region chunk of 8 Megabytes before the next allocation occurs), or the reference is freed in the case of straightforward `ByteBuffer` allocation.

Figure 8 shows a comparison of allocation times. The first thing to note is, that all the buddy-based schemes are dramatically better than relying on the JVM's management of `ByteBuffer`. This essentially means that without a buffer pooling mechanism, creation of intermediate buffers for sending or receiving messages in a Java messaging system can have detrimental effect on the performance. Results are averaged over many repeats, and the overhead of garbage collection cycles will be included in the results in an averaged sense; this is a fair representation of what will happen in a real application. In a general way we attribute the dramatic increase in average allocation time for large `ByteBuffer`s as due to forcing proportionately many garbage collection cycles. All the buddy variants (by design) avoid this overhead. The best strategy in almost all cases is Buddy1 using direct buffers.

Qualitative measurements of memory footprint suggest the current implementation of Buddy2 also has about a 20% bigger footprint because of the extra objects stored. The final version of this paper will include memory footprints comparison of each approach.

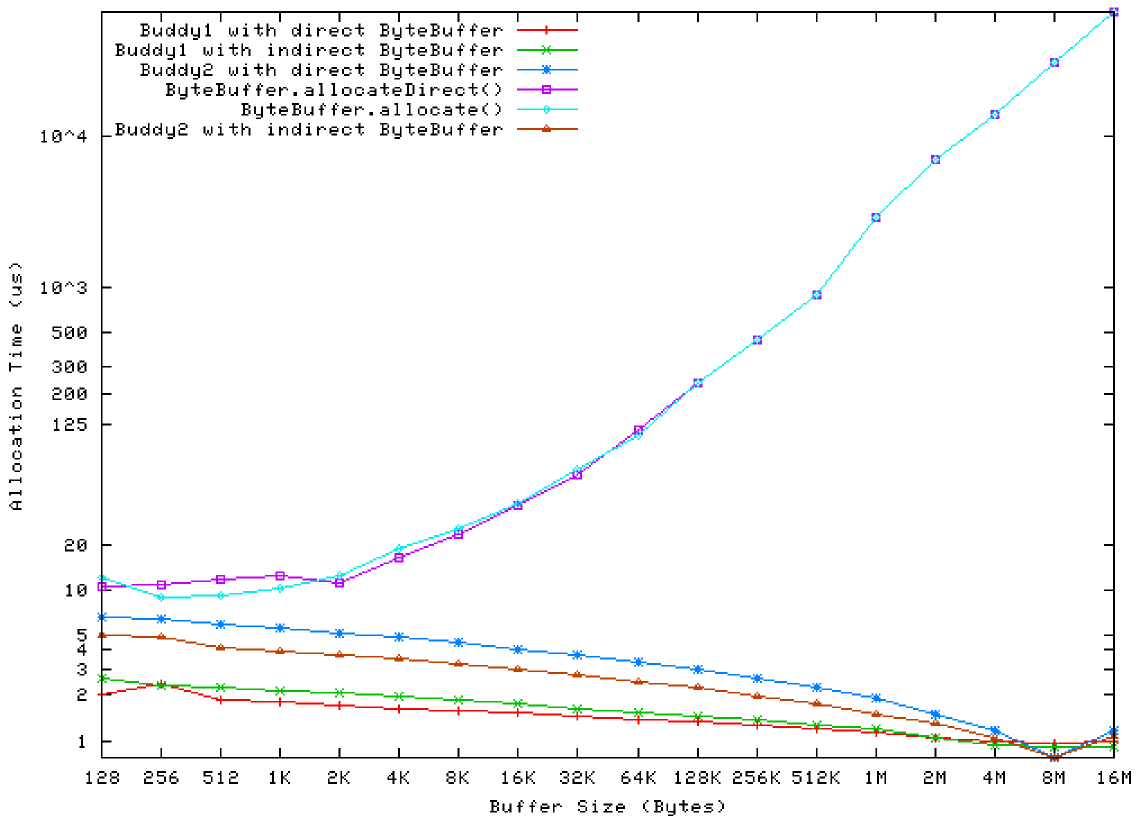
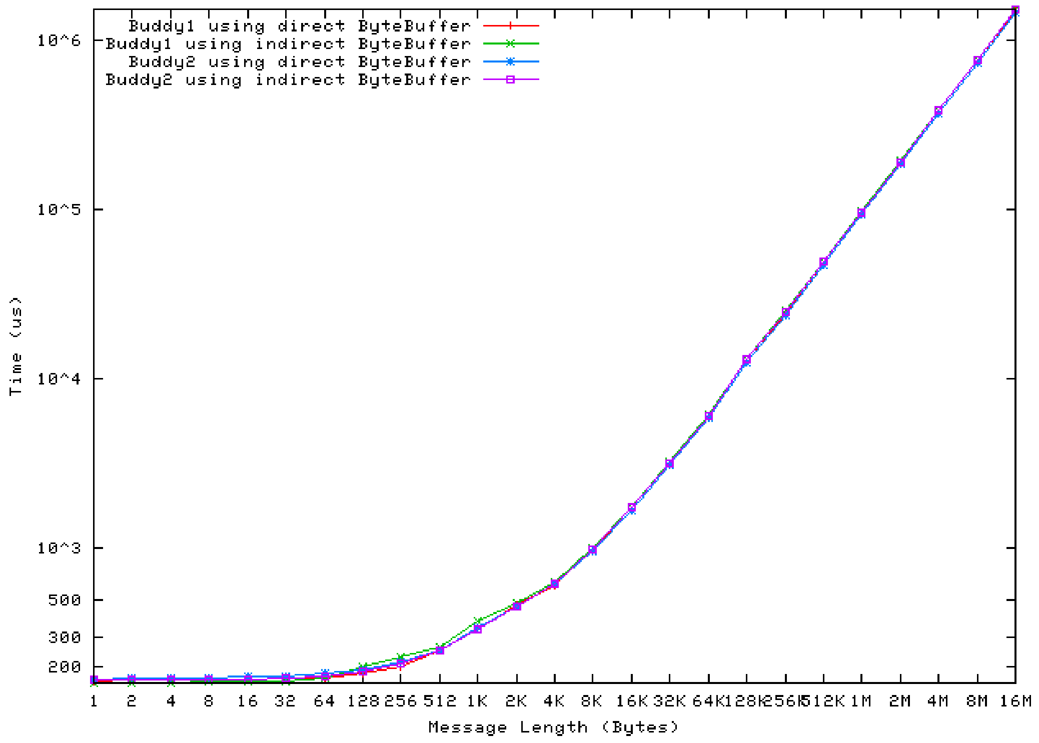


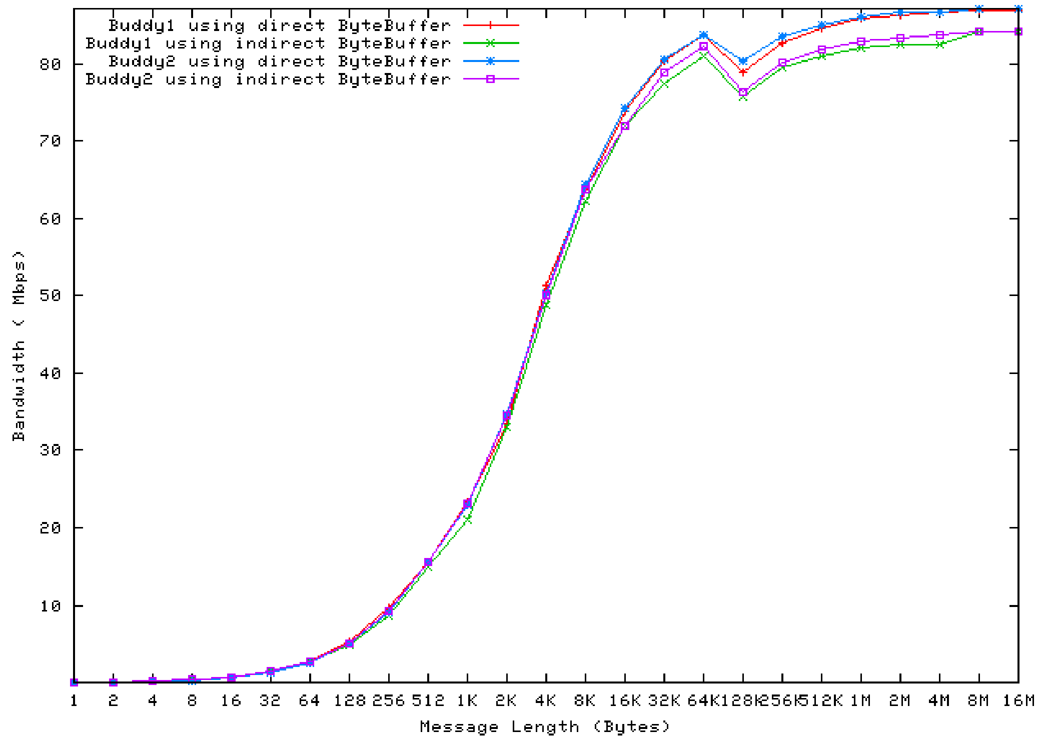
Figure 8: Allocation Time Comparison

4.2 Incorporating Buffering Strategies into MPJE

In this test, we compare transfer times and throughput measured by a simple ping-pong benchmark using each of the different buffering strategies. These tests were performed on Fast Ethernet. We refer interested readers to Appendix 2 for a detailed discussion on our methodology, test environment and comparison of MPJE against other messaging systems. The reason for performing this test is to see if there are any performance benefits for using direct `ByteBuffer`s. From the source-code of the NIO package, it appears that the JVM maintains a pool of direct `ByteBuffer`s for internal purposes. These buffers are used for reading and writing messages into the socket. A user provides an argument to `SocketChannel`'s write or read method. If this buffer is direct, it is used for writing or reading messages. If this buffer is indirect, a direct byte buffer is acquired from direct byte buffer pool and the message is copied first before writing or reading it into the socket. Thus, we expect to see an overhead of this additional copying for indirect buffers.



a. Transfer Time Comparison on Fast Ethenet



b. Throughput Comparison on Fast Ethernet

Figure 9: Latency and Throughput Comparison

Figure 9a shows transfer time comparison. Normally transfer time comparison is useful for evaluating the performance of smaller messages. We do not see any significant performance difference for small messages. On the other hand, Figure 9b shows a throughput comparison and it shows MPJE achieves maximum throughput when using direct buffer in combination with either of the buddy implementations. We expect to see this performance overhead related to indirect buffers to be more significant for faster networks like Gigabit Ethernet and Myrinet. The drop in throughput at 128Kbytes message size is because of the change in communication protocol from eager send to rendezvous.

5. Conclusions and Future work

In this paper, we have discussed the design and implementation of our buffering layer, which uses our own implementation of buddy algorithm for buffer pooling. For a Java messaging system, it is useful to rely on application level memory management technique instead of relying on Java's garbage collector because constant creation and destruction of buffers can be a costly operation. We benchmarked our two pooling mechanisms against each other using combinations of direct and indirect byte buffers. We found that one of the pooling strategies (`Buddy1`) is faster than the other with a smaller memory footprint. Also, we demonstrated the performance gain of using direct byte buffers.

We released a beta version of our software in early September 2005. This release contains our buffering API with two implementations of buddy allocation scheme. This API is self-contained and can be used by other Java applications for application level memory management. Currently, we are working to release mpiJava as a native MPI device for MPJE. Also, we have implemented initial prototypes of `gmdev` and `smpdev`. Once these devices are stable and optimised, we aim to release them as part of our software.

References

[1] M. Baker, H. Ong, A. Shafi, A Status Report: Early Experiences with the implementation of a Message Passing System using Java NIO, DSG Technical Report DSGTR06102004, October 2004, http://dsg.port.ac.uk/projects/mpj/docs/res/DSGTR06102004.pdf

[2] M. Welsh and D. Culler, Jaguar: Enabling Efficient Communication and I/O in Java, Concurrency: Practice and Experience, Special Issue on Java for High-Performance Applications Vol. 12, pp. 519-538, December 1999.

[3] C.-C. Chang and T. von Eicken, Javia: A java interface to the virtual interface architecture, Concurrency: Practice and Experience, Special Issue on Java for High-Performance Applications, Dec. 1999.

[4] Myricom, The MX (Myrinet eXpress) library, http://www.myri.com

[5] MPJ Express, http://dsg.port.ac.uk/projects/mpj

[6] M. Baker, B. Carpenter, G. Fox, S. Hoon Ko, and S. Lim. mpiJava: An Object-Oriented Java interface to MPI. Presented at International Workshop on Java for Parallel and Distributed Computing, IPPS/SPDP 1999, San Juan, Puerto Rico, April 1999.

[7] S. B. Lim, B. Carpenter, G. Fox, and H. Lee, A Device Level Communication Library for the HPJava Programming Language. In proceedings of the IASTED International Conference on Parallel and Distributed Computing and Systems (PDCS 2003), November 2003

[8] M. Baker. B. Carpenter, A. Shafi, MPJ: Enabling Parallel Simulations in Java, DSG Technical Report DSGTR19062005, June 2005, http://dsg.port.ac.uk/projects/mpj/docs/res/DSGTR19062005.pdf

[9] B. Carpenter, G. Fox, S. Ko, and S.Lim. mpiJava 1.2: API Specification, October 1999, http://www.npac.syr.edu/projects/pcrc/mpiJava/mpiJava.html

[10] B. Carpenter, V. Getov, G. Judd, A. Skjellum, G. Fox: MPJ: MPI-like message passing for Java, Concurrency-Practice and Experience 12(11): 1019-1038 (2000)

[11] The Message Passing Interface (MPI) standard, http://www-unix.mcs.anl.gov/mpi

[12] D. E. Knuth, The Art of Computer Programming, Volume 1: Fundamental Algorithms, Second Edition, Reading, Massachusetts, USA, Addison Wesley, 1973.

[13] The Java Native Interface (JNI) Specifications, http://java.sun.com/j2se/1.5.0/docs/guide/jni/

[14] The Java New I/O Specifications, http://java.sun.com/j2se/1.5.0/docs/guide/nio/

[15] The Java Service Wrapper Project, http://sourceforge.net/projects/wrapper/

[16] D. Turner, A. Oline, X. Chen, and T. Benjegerdes, Integrating New Capabilities into NetPIPE, Euro PVM/MPI, Venice, Italy, September 2003

Appendix 1: The MPJ Express Software

MPJ Express is an implementation of the MPI bindings for Java. In this section, we will briefly cover the high-level features of MPJE.

MPJ Express Design

MPJE has a layered design that allows incremental development and provides the capability to swap communication layers in or out as needed. To achieve this, the design incorporates two device levels; MPJ device (`mpjdev`) [7] and `xdev` [8]. In this paper, we limit our discussion to only one implementation of xdev called `niodev`, which is based on the Java NIO package. We are working on other implementations of xdev—`smpdev` for shared memory communications and `gmdev` for Myrinet. More details about the design can be found in [8].

Point-to-Point Communications

MPJE provides blocking and non-blocking point-to-point communications that can be used to send arrays of basic Java datatypes as well as objects. MPJE provides four send modes, which have been defined in the MPI specifications document [11]. These control the semantics of internal buffering. The standard send mode uses eager-send for message sizes up to 128 Kbytes and rendezvous protocol for messages larger than this size. Synchronous send mode uses the rendezvous protocol for communications. The ready send mode is a wrapper to standard send[2]. The last one, called buffered-send follows the model implementation of buffered mode outlined in the MPI specifications document.

Communicators, Groups and Contexts

MPI provides higher-level abstractions to create parallel libraries, which include communicators, groups, and contexts. Communicators along with groups provide process naming; each process in MPI is identified by its rank. The context, which is an attribute of a communicator, provides a safe communication universe – it can be thought of an additional tag on the messages. MPI provides two types of communicators. The first is intracommunicators, which provides communications within a group. The second is intercommunicators, which provides communications between two groups. We have implemented these features at the exported API level of MPJ Express.

[2] The standard mode of send is a valid implementation of ready send.

Virtual Topologies and Collective Communications

In compliance with the MPI specifications, MPJE provide two types of virtual topologies; cartesian and graph. The cartesian topology especially is extensively used in scientific applications. Also, we have implemented all collective communication methods; most of these are implemented on top of point-to-point communications. Two of the widely used methods, broadcast and barrier use n-ary tree structure.

The Runtime Infrastructure

An important subsystem of a Java messaging system is the mechanism for bootstrapping MPJE processes across various platforms. It is a challenge to make this mechanism portable. If the compute-nodes are running a UNIX-based OS, it is possible to remotely execute commands using RSH/SSH. But if the compute-nodes are running Windows, these utilities are not universally available. The MPJE runtime provides a unified way of starting MPJE processes on compute-nodes irrespective of the operating system. The runtime system consists of two modules. The `daemon` module executes on compute-nodes and listens for requests to start MPJE processes. The daemon is a Java application listening on a IP port, which starts a new JVM whenever there is a request to execute an MPJE process. The `mpjrun` module acts as a client to the daemon module. This module may be invoked on, for example, on the cluster head-node. It will contact daemons, which will start MPJE processes in a new JVM.

Java allows one to run applications using class files as open in directory structure, or class files bundled as a JAR file. The MPJE runtime allows the execution of MPJE applications in either format. Users may want to load MPJE JARs and classes either remotely or locally on the compute-nodes as shown in Figure 10. With the remote loader, it is possible to load all classes (application and MPJE code) from the user's development node, such as a cluster head-node in Figure 10b. This is useful in scenarios when there is no shared file system and the code is constantly being modified at the head-node. With local loader, it is possible to load all classes (application and MPJ Express code) from the compute-node as in Figure 10a. This might be useful if there is a shared file system. As all classes are loaded locally, this might provide better performance in comparison to remote loader.

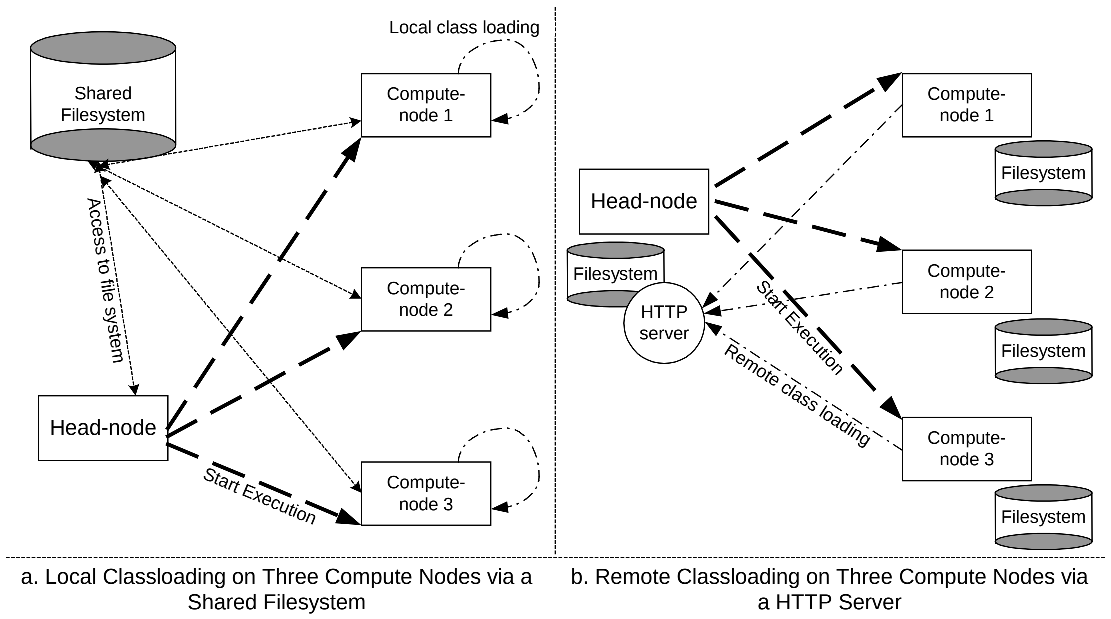


Figure 10: Demonstration of Local and Remote Loading

MPJE uses the Java Service Wrapper Project [15] software to install daemons as a native OS service. This essentially means that there is some platform specific code used in order to achieve this.

Appendix 2: Benchmarking MPJE Against Other Messaging Systems

In this section we present latency and throughput comparison of MPJE against MPICH, LAM/MPI and mpiJava using a ping-pong benchmark. The test environment for collecting the performance results consists of a cluster of 8 dual Intel Xeon 2.8 GHz PCs using the Intel E7501 chipset. The PCs were equipped with 2 Gigabytes 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 Debian GNU/Linux with the 2.4.30 Linux kernel. The software used for the Intel Gigabit adaptor was the proprietary Intel e-1000 device driver. The JDK version that has been used for these tests for mpiJava and MPJE is Sun J2SE 1.5 (update 4). MPICH and LAM/MPI have been compiled with GNU GCC 3.3.5.

Calculating Latency

We define latency as the time taken to transfer one byte message. While calculating latency for MPJE, mpiJava etc., we noticed variability in timing measurements especially for small messages. This variance is clearly noticeable not only for MPI libraries, but also for simple C or Java ping-pong benchmarks like Netpipe [16]. The latency for C varies from 64 microseconds to 128 microseconds. Also, the latency of Java varies from 64 microseconds, 125 microseconds, and 250 microseconds. To show the variance, we have plotted histograms shown in Figure 11. The x-axis shows the latency in microseconds. The y-axis is a measure of how many times this particular latency value has been repeated. The longer the bar, the more repetitions are associated with this latency value. Figure 11a shows the histograms of Java ping-pong over 10000 iterations for 1 Byte message size. This shows the latency measurement is jumping between 64 microseconds, 125 microseconds and 250 microseconds. Figure 11b shows the histograms for C ping-pong experiments for 1 byte message over 10000 iterations. It is also varying between 64 microseconds, 125 microseconds, and 250 microseconds.

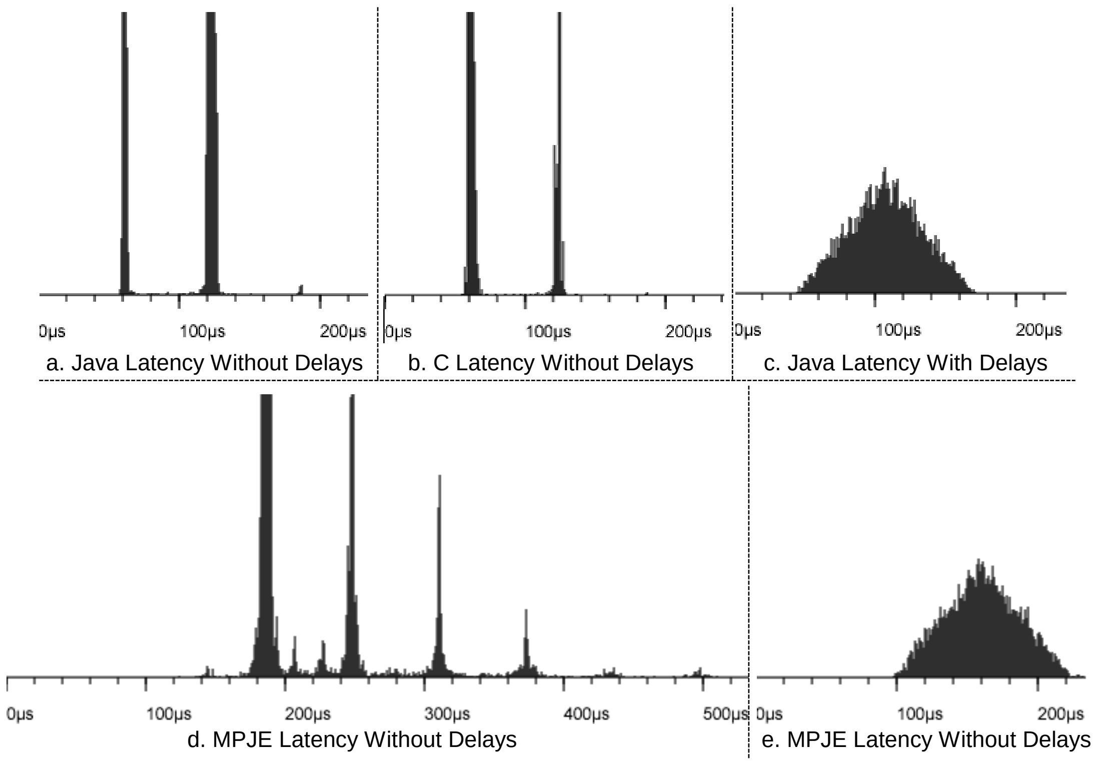


Figure 11: Latency Calculation Histograms

Eventually, we found out that network card drivers on our cluster have 64 microseconds network latency, which is the main cause of variance of latency measurements. The network latency of network card drivers is an attribute that determines the polling interval for checking new messages. A network latency of 64 microseconds means that after every 64 microseconds, the network card driver checks for new messages. We verified the cause for variance by running the same benchmarks on another cluster, with latency in the orders of nanoseconds.

To negate the affect of network card's latency, we introduced random delays before sending the message back to the sender. To explain this, imagine we are calculating latency between two machines A and B. Machine A starts the timer, sends a message to B, machine B receives this message, and sends it back to machine A. When machine A receives this message, it stops the timer. The duration that machine A has calculated is the round trip time for sending a message from machine A to machine B.

In our modified technique to calculate round trip time, we introduce random delays at machine B before sending the message back to machine A. We subtract this random delay from the round trip time calculated by machine A. Using this technique; we were able to negate the affect of network card latency. Figure 11c shows the histograms of Java ping-pong benchmark using this technique.

The latency histograms shown in Figure 11c and Figure 11e looks like a pyramid. Let `p` be physical transfer time, and `t` be the network latency of the card. The minimum of this pyramid is `p`, peak exists at `t+p`, and maximum will be `t+2p`. The peak also represents average latency. Using this technique, the average latency for C is 83 microseconds and Java is 109 microseconds. As can be noticed, the overhead incurred by the JVM is 26 microseconds. Figure 11d shows the histograms for MPJE without delays and Figure 11e shows the histograms for MPJE after introducing random delays.

Transfer Time and Throughput on Fast Ethernet

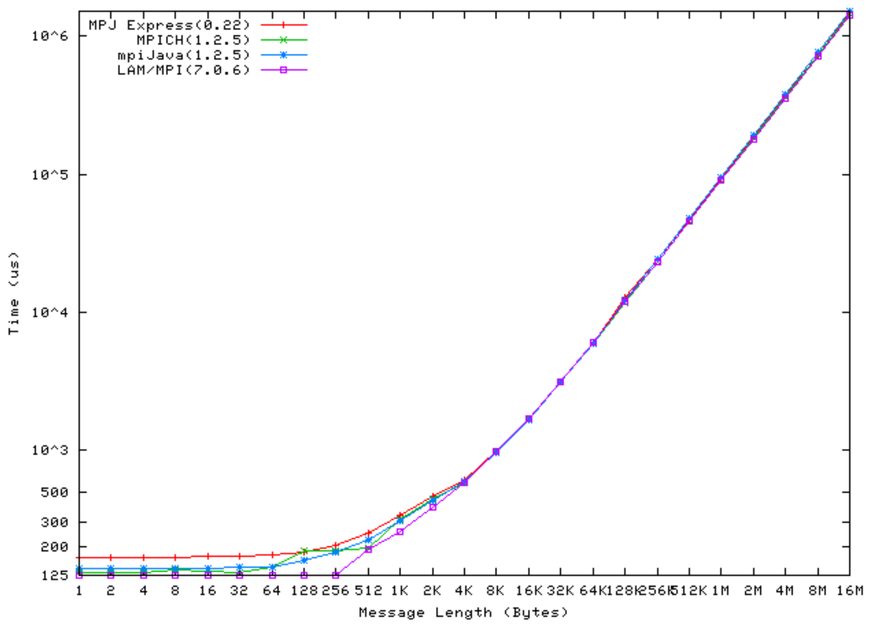
In this section, we present transfer time and throughput comparison of MPJE against other messaging systems on Fast Ethernet. The latency for MPJE is 161 microseconds, mpiJava is 145 microseconds, MPICH is 126 microseconds, and LAM/MPI is 125 microseconds as shown in Figure 12a. The reason for higher latency is the additional layer (JVM) introduced by the Java language. We noted that the difference between the latency of C and Java is 26 microseconds. This difference is the main cause of overhead of MPJE.

Figure 12b shows the throughput comparisons of MPJE against other MPI libraries. The maximum bandwidth achieved by MPJE is 87 Mbps, mpiJava is 83 Mbps, MPICH is 88 Mbps, and LAM/MPI is 90 Mbps. MPJE performs better than mpiJava for large messages because of the JNI overhead [2], which is significant for large messages. The drop at 128K is the change of communication protocols from eager send to rendezvous.

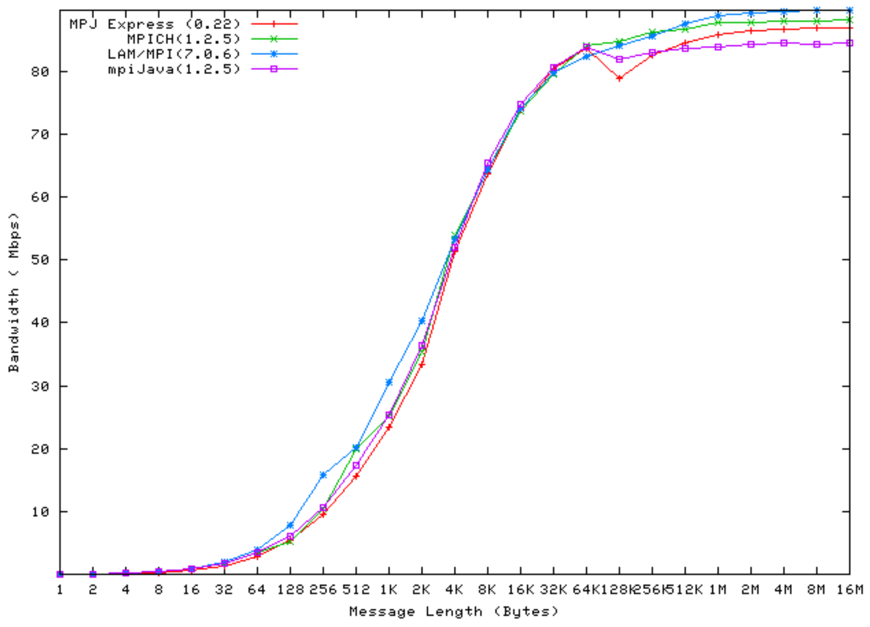
Transfer Time and Throughput on Gigabit Ethernet

In this section, we present transfer time and throughput comparison of MPJE against other messaging systems on Gigabit Ethernet. The latency for MPJE is 131 microseconds and mpiJava is 101 microseconds as shown in Figure 13a. The reason for higher latency of MPJE is the additional layer (JVM) introduced by the Java language.

Figure 13b shows the throughput comparisons of MPJE against other MPI libraries. The maximum bandwidth achieved by MPJE is 680 Mbps and mpiJava is 597 Mbps. The drop at 128K is the change of communication protocols from eager send to rendezvous. The value of this protocol switch limit can be configured and it appears that the value of 128K is not optimal for GigE. On the other hand, rendezvous protocol reduces an additional copying and memory requirements for low-level communication device.

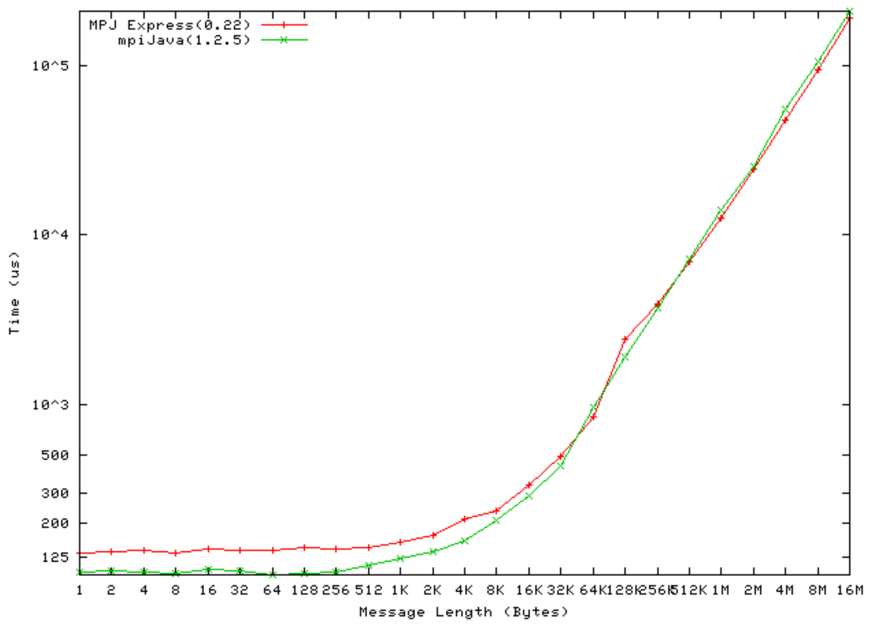


a. Latency Comparison on Fast Ethernet

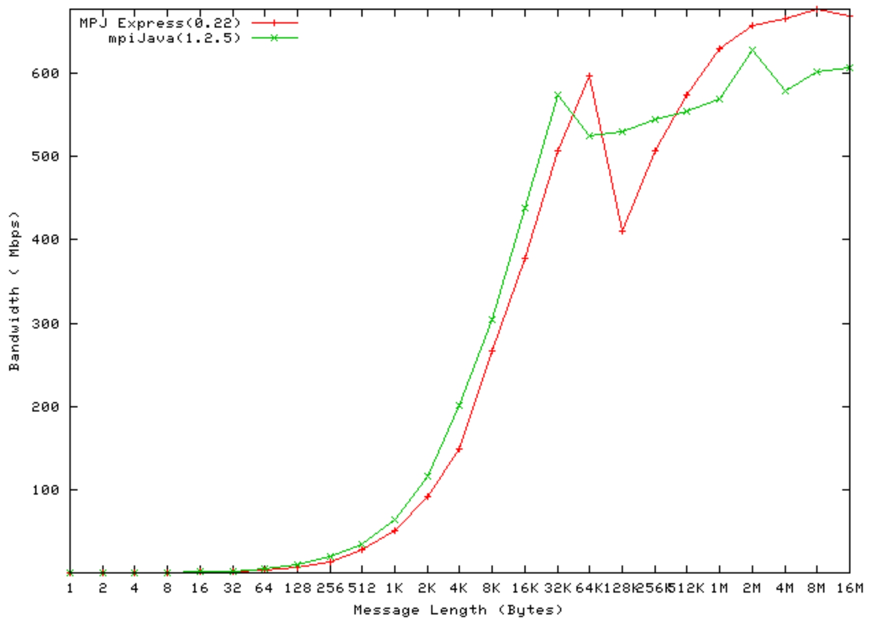


b. Throughput Comparison on Fast Ethernet

Figure 12: Comparison of MPJE Against mpiJava, MPICH, and LAM/MPI on Fast Ethernet



a. Latency Comparison on Gigabit Ethernet



b. Throughput Comparison on Gigabit Ethernet

Figure 13: Comparison of MPJE Against mpiJava on Gigabit Ethernet