Scalable Computing: Practice and Experience
Volume 8, Number 4, pp. 343–358. http://www.scpe.org

ISSN 1895-1767
© 2007 SWPS

A BUFFERING LAYER TO SUPPORT DERIVED TYPES AND PROPRIETARY NETWORKS FOR JAVA HPC*

MARK BAKER†, BRYAN CARPENTER‡, AND AAMIR SHAFI§

Abstract. MPJ Express is our implementation of MPI-like bindings for Java. In this paper we discuss our intermediate buffering layer that makes use of the so-called direct byte buffers introduced in the Java New I/O package. The purpose of this layer is to support the implementation of derived datatypes. MPJ Express is the first Java messaging library that implements this feature using pure Java. In addition, this buffering layer allows efficient implementation of communication devices based on proprietary networks such as Myrinet. In this paper we evaluate the performance of our buffering layer and demonstrate the usefulness of direct byte buffers. Also, we evaluate the performance of MPJ Express against other messaging systems using Myrinet and show that our buffering layer has made it possible to avoid the overheads suffered by other Java systems such as mpiJava that relies on the Java Native Interface.

Key words. Java, MPI, MPJ express, MPJ, mpiJava

1. Introduction. The challenges of making parallel hardware usable have, over the years, stimulated the introduction of many novel languages, language extensions, and programming tools. Lately though, practical parallel computing has mainly adopted conventional (sequential) languages, with programs developed in relatively conventional programming environments usually supplemented by libraries such as MPI that support a parallel programming paradigm. This is largely a matter of economics: creating entirely novel development environments matching the standards programmers expect today is expensive, and contemporary parallel architectures predominately use commodity microprocessors that can best be exploited by off-the-shelf compilers.

This argues that if we want to "raise the level" of parallel programming, one practical approach is to move towards advanced commodity languages. Compared with C or Fortran, the advantages of the Java programming language include higher-level programming concepts, improved compile-time and run-time checking, and as a result, faster problem detection and debugging. Its "write once, run anywhere" philosophy allows Java applications to be executed on almost all popular platforms. It also supports multi-threading and provides simple primitives like `wait()` and `notify()` that can be used to synchronize access to shared resources. Recent Java Development Kits (JDKs) provide greater functionality in this area, including semaphores and atomic variables. In addition, Java's automatic garbage collection, when exploited carefully, relieves the programmer of many of the pitfalls of lower-level languages. During the early days of Java, it was criticized for its poor performance [4]. The main reason was that Java executed as an interpreted language. The situation has improved with the introduction of Just-In-Time (JIT) compilers, which translate bytecode into the native machine code that then gets executed.

MPJ Express [14] is a thread safe Java HPC communication library and runtime system that provides a high quality implementation of the mpiJava 1.2 [6] bindings—an MPI-like API for Java. An important goal of our messaging system is to implement higher MPI [15] abstractions including derived datatypes in pure Java. In addition, we note the emergence of low-latency and high-bandwidth proprietary networks that have had a big impact on modern messaging libraries. In the presence of such networks, it is not *practical* to only use pure Java for communication. To tackle these issues of supporting derived datatypes and proprietary networks, we provide an intermediate buffering layer in MPJ Express. Providing an efficient implementation layer is a challenging aspect of a Java HPC messaging software. The low-level communication devices 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 socket's `write()` method. For proprietary networks like Myrinet [16], this memory region can be registered for Direct Memory Access (DMA) transfers, or copied to a DMA

*The authors would like to thank University of Portsmouth for supporting this research. The research work presented in this paper was conducted by the authors, as part of the Distributed Systems Group, at the University of Portsmouth.

†School of Systems Engineering, University of Reading, Reading, RG6 6AY, UK (mark.baker@computer.org)

‡Open Middleware Infrastructure Institute, University of Southampton, Southampton, SO17 1BJ, UK (dbc@ecs.soton.ac.uk)

§Center for High Performance Scientific Computing, NUST Institute of Information Technology, Rawalpindi, Pakistan (aamir.shafi@niit.edu.pk)

capable part of memory and sent using low level Myrinet communication methods. Until quite recently doing this kind of thing in Java was difficult.

The JDK 1.4 introduced the Java New I/O (NIO) [11] package. In NIO, read and write methods on files and sockets (for example) are mediated through a family of buffer classes handled by the Java Virtual Machine (JVM). The underlying `ByteBuffer` class essentially implements an array of bytes, but in such a way that the storage can be outside the JVM heap (so called *direct* byte buffers).

So now if a user of a Java messaging system sends an array of ten integers, they can be copied to a `ByteBuffer`, which is used as an argument to the `SocketChannel`'s `write()` method. Similary, if the user intends to communicate derived datatypes, the individual basic datatype elements of this derived type can be packed onto a contiguous `ByteBuffer`. The higher and lower levels of the software can use generic functionality provided by a buffering layer to communicate both basic and advanced datatypes, including Java objects and derived types. For proprietary networks like Myrinet, NIO provides a viable option because it is now possible to get memory addresses of direct byte buffers, which can be used to register memory regions for DMA transfers. Using direct buffers may eliminate the overhead [18] incurred by additional copying when using the Java Native Interface (JNI) [9]. On the other hand, it may be preferable to create a native buffer using JNI. These buffers can be useful for a native MPI or a proprietary network device.

For these reasons, we have designed an extensible buffering layer that allows various implementations based on different storage mediums, such as direct or indirect `ByteBuffer`s, byte arrays, or memory allocated in the native C code. The higher levels of MPJ Express 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 creation time of these buffers can affect the overall communication time, especially for large messages. Our buffering strategy uses a pooling mechanism to avoid creating a buffer instance for each communication method. Our current implementation is based on Knuth's buddy algorithm [12], but it is possible to use other pooling techniques.

A closely related buffering API with similar gather and scatter functionality was originally introduced for Java in the context of the Adlib communication library used by HPJava [13]. In our current work, we have extended this API to support the derived datatypes in a fully functional MPI interface.

The main contribution of this paper is the in-depth analysis of the design and implementation of our buffering layer that allows high performance communication and supports implementing derived datatypes at the higher level. MPJ Express is the first Java messaging library that supports derived datatypes using pure Java. In addition, we have evaluated the performance of MPJ Express on Myrinet—a popular high performance interconnect. Also, we demonstrate the usefulness of direct byte buffers in Java messaging systems.

The remainder of this paper is organized as follows. Section 2 discusses the details of the MPJ Express buffering layer. Section 3 describes the implementation of derived datatypes in MPJ Express. In Section 4, we evaluate the performance of our buffering strategies, this is followed by a comparison of MPJ Express against other messaging systems on Myrinet. We conclude the paper and discuss future research work in Section 5.

1.1. Related Work. Under the general umbrella of exploiting Java in "high level" parallel programming, there are environments for *skeleton-based* parallel programming that are implemented in Java, or support Java programming. These include *muskel* [8] and *Lithium* [1]. At the present time muskel appears to be focussed on a coarse grained data flow style of parallelism, rather than the sort of Single Program Multiple Data (SPMD) parallelism addressed by MPI-like systems such as MPJ Express. Lithium encompasses SPMD parallelism through its *map* skeleton. So far as we can tell, Lithium is agnostic about how the processes in the "map" communicate amongst themselves, and in principle we see no reason why they could not use MPJ Express for this purpose. To this extent Lithium and our approach could be seen as complementary.

In a similar vein, Alt and Gortlatch [2] developed a prototype system for Grid programming using Java and RMI. Again the issues addressed in their work are somewhat orthogonal from the concerns of the present paper. But it is possible that some of their ideas for discovery of parallel compute hosts could be exploited by a future version of MPJ Express. Such approaches might supercede what we call the *runtime system* of the present MPJ Express—responsible for initiating node tasks on remote hosts.

In another related strand of research, there are systems that provide Java implementations of Valiant's Bulk Synchronous Parallel computing model (BSP). These include JBSP [10] and PUBWCL [5]. In the sense that these are providing a messaging platform to essentially do data parallel programming in Java, they compete more

directly with MPJ Express. They are distinguished from our work in focussing on a more specific programming model. MPI-based approaches embrace a significantly different, and in some respects wider, class of parallel programming models (on some platforms one could, of course, sensibly implement BSP in terms of MPI).

mpiJava [3] is a Java messaging system that uses JNI to interact with the underlying native 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 may be required between the JVM and the native C library. This overhead is especially noticeable for large messages, if the JVM does not support pinning of memory.

Javia [7] 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 99% of the raw hardware.

An effort similar to Javia is Jaguar [18]. 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 general and efficient buffering layer that can be used for pure Java and proprietary devices in Java messaging systems alike.

2. The Buffering Layer in MPJ Express. In this section, we discuss our approach to designing and implementing our MPJ Express buffering layer that is 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 even more complex. Most of the complex operations used at the higher levels of the library, such as communicating objects and gather or scatter operations, are also supported by this buffering layer.

Before we go into the details of the buffering layer implementation, it is important to consider how this API is used. The higher-level of MPJ Express, specifically the point-to-point send methods, *pack* the user message onto a `mpjbuf` buffer. Once the user data, which may be primitive datatypes or Java objects, has been packed onto a `mpjbuf` buffer, the reference of this buffer is passed to lower communication devices that communicate data from the static and dynamic storage structures. At the receiving side the communication devices receive data into a `mpjbuf` buffer storing them in static and dynamic storage structures. Once the message has been received, the point-to-point receive methods *unpack* the `mpjbuf` buffer data onto user specified arrays.

2.1. The Layout of Buffers. An `mpjbuf` buffer object contains two data storage structures. The first is a *static* storage structure, in which the underlying storage primitive is an implementation of the `RawBuffer` interface. An implementation of the static storage structure, called `NIOBuffer`, uses direct or indirect `ByteBuffer`s. The second is a *dynamic* storage structure where a byte array is the storage primitive. The static portion of the `mpjbuf` buffer has predefined size, and can contain only primitive datatypes. The dynamic portion of the `mpjbuf` buffer is used to store serialized Java objects, where it is not possible to determine the size of the serialized objects beforehand.

A message consists of zero or more sections stored physically on the static or dynamic storage structure. 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 the value `java.nio.ByteOrder.BIG_ENDIAN` or `java.nio.ByteOrder.LITTLE_ENDIAN`. The encoding property of a newly created buffer is determined by the return value of the method `java.nio.ByteOrder.nativeOrder()`. A developer may change the format to match the encoding property of the underlying hardware, which results in efficient numeric representation at the JVM layer.

As shown in Figure 2.1, 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 a value of 8. The general layout of an individual section stored on the static buffer is shown in Figure 2.2.

Figure 2.2 shows that the length of a message header is 8 bytes. The value of the first byte defines the elements type contained in the section. The possible values for static and dynamic buffers are listed in Table 2.1

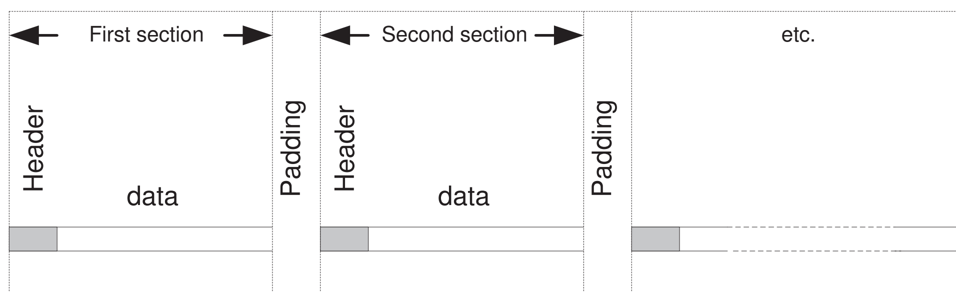

Fig. 2.1. *The Layout of a Static Storage Buffer*

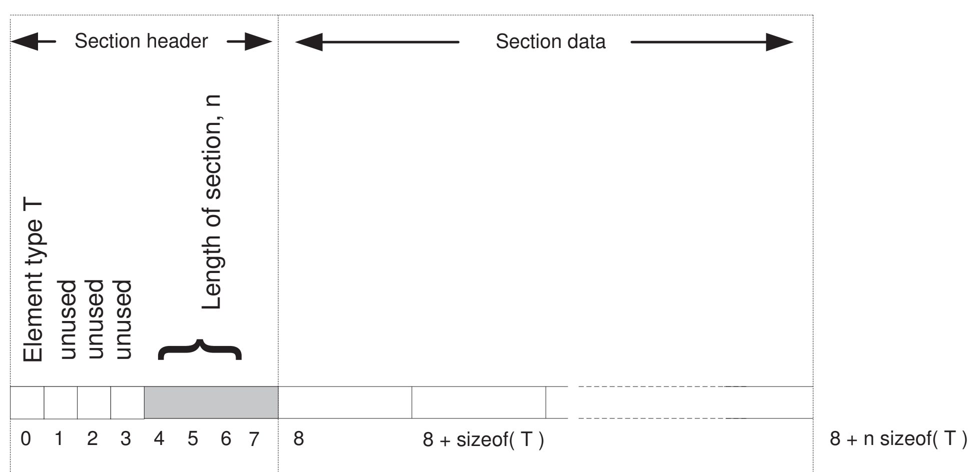

Fig. 2.2. *The Layout of a Single Section*

and Table 2.2, respectively. The next three bytes are not currently used, and reserved for possible future use. The following four bytes contain the number of elements contained in this section, i. e. the section length. This numerical value is represented according to the encoding property of the buffer. The size of the header in bytes is SECTION_OVERHEAD, which has a value of 8. If the section is static, the header is followed by the values of the elements, again represented according to the encoding property of the buffer. If the section is dynamic, the "Section data" is absent from Figure 2.2 because the data is in the dynamic buffer which is a byte array. The Java serialization classes (`java.io.ObjectOutputStream` and `java.io.ObjectInputStream`) dictate the format of the dynamic buffer.

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 write mode. Similarly, it is not permitted to write to a buffer when it is in read mode.

2.2. The Buffering API. The most important class of the package used for packing and unpacking data is `mpjbuf.Buffer`. This class provides two storage options: static and dynamic. Implementations of static storage use the interface `mpjbuf.RawBuffer`. It is possible to have alternative implementations of static section depending on the actual raw storage medium. In addition, it also contains an attribute of type `byte[]` that represents the dynamic section of the message. Figure 2.3 shows two implementations of the `mpjbuf.RawBuffer` interface. The first, `mpjbuf.NIOBuffer` is an implementation based on `ByteBuffer`s. The second, `mpjbuf.NativeBuffer` is an implementation for the native MPI device, which allocates memory in the native C code. Figure 2.3 shows the primary buffering classes in the mpjbuf API.

The higher and lower levels of MPJ Express use only a few of methods provided by the `mpjbuf.Buffer` class to pack and unpack message data. In addition, the class also provides some utility methods. Some of the main functions are shown in Figure 2.4. Note that identifier `type` used in the figure represents all Java basic datatypes and objects.

Table 2.1
Datatypes Supported by a Static Buffer

Datatype	Corresponding Values
Integer	`mpjbuf.Type.INT`
Byte	`mpjbuf.Type.BYTE`
Short	`mpjbuf.Type.SHORT`
Boolean	`mpjbuf.Type.BOOLEAN`
Long	`mpjbuf.Type.LONG`
Float	`mpjbuf.Type.FLOAT`
Double	`mpjbuf.Type.DOUBLE`

Table 2.2
Datatypes Supported by a Dynamic Buffer

Datatype	Corresponding Values
Java objects	`mpjbuf.Type.OBJECT`
Integer	`mpjbuf.Type.INT_DYNAMIC`
Byte	`mpjbuf.Type.BYTE_DYNAMIC`
Short	`mpjbuf.Type.SHORT_DYNAMIC`
Boolean	`mpjbuf.Type.BOOLEAN_DYNAMIC`
Long	`mpjbuf.Type.LONG_DYNAMIC`
Float	`mpjbuf.Type.FLOAT_DYNAMIC`
Double	`mpjbuf.Type.DOUBLE_DYNAMIC`

The `write()` and `read()` methods shown in Figure 2.4 are used to write and read contiguous Java arrays of all the primitive datatypes including object arrays. The `write()` method copies `numEls` values of the `src` array starting from `srcOff` onto the buffer. Conversely, the `read()` method copies `numEls` values from the buffer and writes them onto `dest` array starting from `srcOff`.

The `gather()` and `scatter()` methods are used to write and read non-contiguous Java arrays of all the primitive datatypes including object arrays. The `gather()` method copies `numEls` values of the `src` array starting from `indexes[idxOff]` to `indexes[idxOff+numEls]` onto the buffer. Conversely, the `scatter()` method copies `numEls` values from the buffer and writes them onto `dest` array starting from `indexes[idxOff]` to `indexes[idxOff+numEls]`.

The `strGather()` and `strScatter()` methods transfer data from or to a subset of elements of a Java array, but in these cases the selected subset is a *multi-strided region* of the array. These are useful operations for dealing with multi-dimensional data structures, which often occur in scientific programming.

To create sections, the `mpjbuf.Buffer` class provides utility methods like `putSectionHeader()`, which takes a datatype as an argument (possible datatypes are shown in Table 2.1 and Table 2.2). This method can only be invoked when the buffer is in a write mode. Once the section header has been created, the data can be copied onto the buffer using the `write()` method for contiguous user data or `gather()` and `strGather()` methods for non-contiguous user data. While the buffer is in read mode, the user can invoke `getSectionHeader()` and `getSectionSize()` methods to read the header information of a message. This is followed by invoking the `read()` method to read data, or `scatter()` and `strScatter()` methods to read non-contiguous data.

The newly created buffer is always in a write mode. In this mode, the user may copy the data to the buffer and then call `commit()`, which puts the buffer in a read mode. The user can now read the data from the buffer and put it back into write mode for any possible future use by calling the `clear()`.

2.3. Memory Management. We have implemented our own application level memory management mechanism based on the buddy allocation scheme [12]. The motivation was to avoid creating an instance of a buffer (`mpjbuf.Buffer`) for every communication operation 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.

Currently the pooling mechanism is specific to direct and indirect `ByteBuffer`s that are used for storing static data when `mpjbuf.NIOBuffer` (an implementation of `mpjbuf.RawBuffer`) is used for static sections.

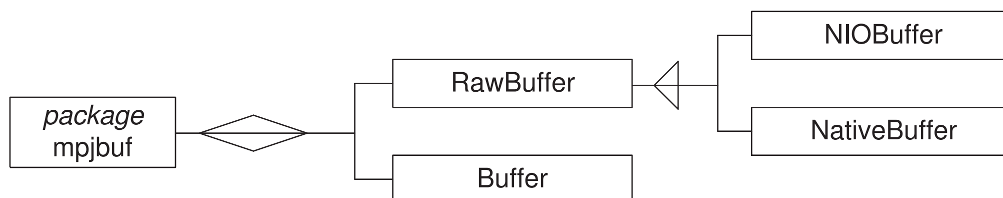

Fig. 2.3. *Primary Buffering Classes in* `mpjbuf`

```
package mpjbuf ;

public class Buffer {
  .. .. ..

  // Write and Read Methods
  public void write(type [] source, int srcOff, int numEls)
  public void read(type [] dest, int dstOff, int numEls)

  // Gather and Scatter Methods
  public void gather(type [] source, int numEls, int idxOff, int [] indexes)
  public void scatter(type [] dest, int numEls, int idxOff, int [] indexes)

  // Strided Gather and Scatter Methods
  public void strGather(type [] source, int srcOff, int rank, int exts,
                                                   int strs, int [] shape)
  public void strScatter(type [] dest, int dstOff, int rank, int exts,
                                                  int strs, int [] shape)

  public void putSectionHeader(Type type)

  public Type getSectionHeader()

  public int getSectionSize()

  public ByteOrder getEncoding()

  public void setEncoding(ByteOrder encoding)

  public void commit()

  public void clear()

  public void free()

  .. .. ..
}
```

Fig. 2.4. *The Functionality Provided by the* `mpjbuf.Buffer` *class*

2.3.1. Review of The Buddy Algorithm. In this section, we will briefly review Knuth's buddy algorithm in the context of MPJ Express. In our implementation, the available memory is divided into a series of buffers. Each buffer has a storage medium associated with it—direct or indirect `ByteBuffer`. Initially, there is no buffer associated with the `BufferFactory`. Whenever a user requests a buffer, the factory checks whether there is a buffer with size greater than the requested size. If a buffer does not exist or does not have free space, a new buffer is created. For managing the buffers, there is a doubly linked list called `FreeList`. This `FreeList` refers to buffers at all possible levels starting from 0 to $\lceil \log_2(REGION_SIZE) \rceil$. The level of a buffer can be thought of an integer, which increases as $\lceil \log_2(s) \rceil$ increases if s is the requested buffer size.

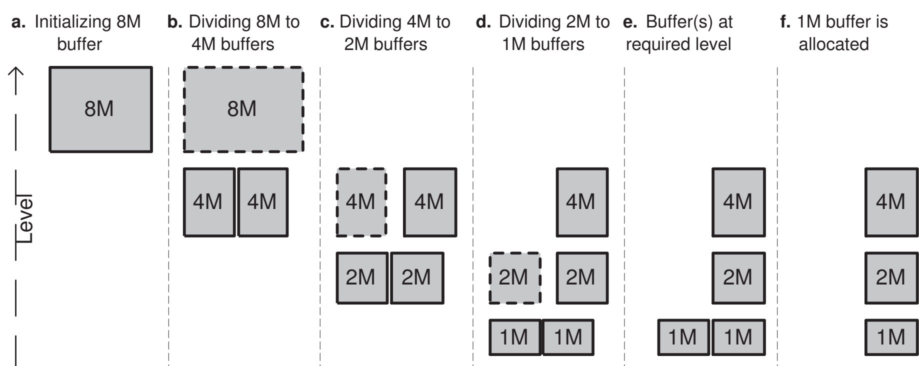

Fig. 2.5. *Allocating a Mbyte Buffer*

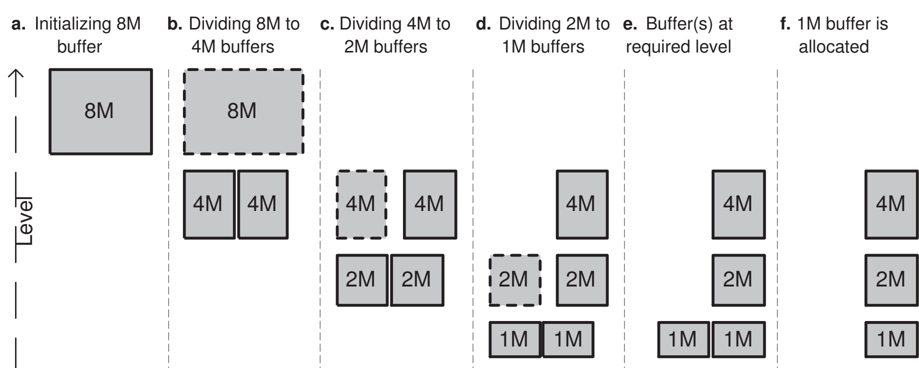

Fig. 2.6. *De-allocating a Mbyte Buffer*

After finding or creating an appropriate buffer 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 its size. Figure 2.5 shows the allocation events for a Mbyte block when the initial region size is 8 Mbytes.

When the buffer is de-allocated, an attempt is made to find the buddy of this buffer. If the buddy is free, the two buffers 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 2.6 shows the de-allocation events when a Mbyte block is returned to the buffer factory.

2.3.2. Two implementations of the Buddy Allocation Scheme for `mpjbuf`. In the MPJ Express 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 `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 `ByteBuffer.slice()` for creating a new byte buffer.

In the buddy algorithm, the region of available storage is conceptually divided into blocks of different levels, hierarchically nested in a binary tree. A free block at level n can be split into two blocks of level $n-1$, half the size. These sibling blocks are called *buddies*. To allocate a number of bytes s, a free block is found and recursively divided into buddies until a block at level $\lceil \log_2(s) \rceil$ is produced. When a block is freed, one checks to see if its buddy is free. If so, buddies are merged (recursively) to consolidate free memory.

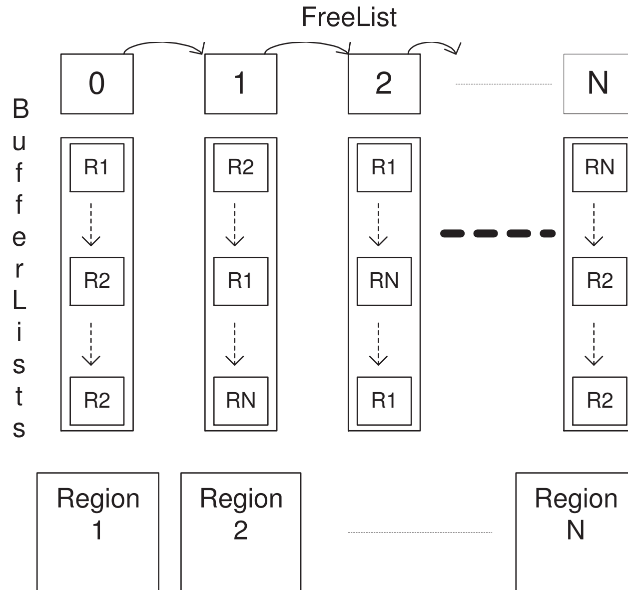

Fig. 2.7. *The First Implementation of Buffer Pooling*

Our first implementation (hereafter called *Buddy1*) is developed with the aim of keeping a small memory footprint for 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 2.7 outlines the implementation details of our first pooling strategy. `FreeList` is a list of `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 $\lceil \log_2(s) \rceil$. Initially, there is no region associated with `FreeList`. An initial chunk of memory of size M is allocated. At this point, `BufferLists` are created starting from 0 to $\log_2(M)$. 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 are 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` grow as new regions are created to match user requests.

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 2.8. While finding an appropriate buffer for a user, this implementation works 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 of this implementation is a bigger memory footprint.

3. The Implementation of Derived Datatypes in MPJ Express. Derived datatypes were introduced in the MPI specification to allow communication of heterogeneous and non-contiguous data. It is possible to achieve some of the same goals by communicating Java objects, but there are concerns about the cost of object serialization—MPJ Express relies on the JDK's default serialization.

Figure 3.1 shows datatype related class hierarchy in MPJ Express. The superclass `Datatype` is an abstract class that is implemented by five classes. The most commonly used implementation is `BasicType` that provides initialization routines for basic datatypes including Java objects.

There are four types of derived datatypes: contiguous, indexed, vector, and struct. Figure 3.1 shows an implementation for each derived datatype including `Contiguous`, `Indexed`, `Vector`, and `Struct`.

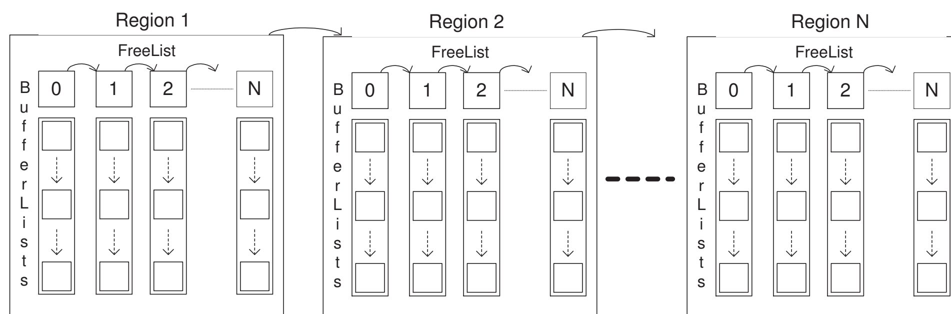

Fig. 2.8. *The Second Implementation of Buffer Pooling*

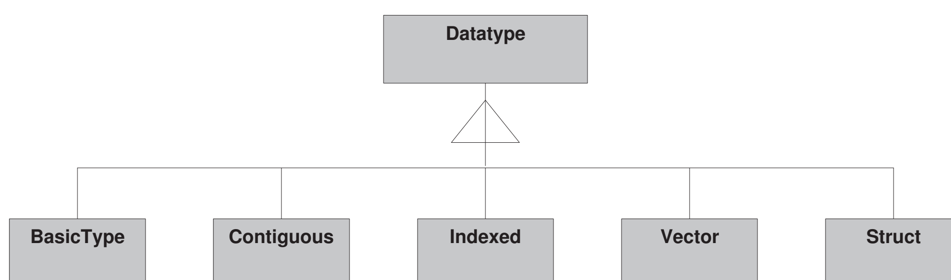

Fig. 3.1. *The Datatype Class Hierarchy in MPJ Express*

The MPJ Express library makes extensive use of the buffering API to implement derived datatypes. Each datatype class contains a specific implementation of the `Packer` interface. The class hierarchy for the implementation of different `Packer`s is shown in Figure 3.2.

Recall that the buffering API provides three kinds of read and write operations. These methods are normally available for use through classes that implement the `Packer` interface. We discuss various implementations that in turn rely on variants of read and write methods provided by the buffering API. The first are the normal `write()` and `read()` methods. The implementation of the `Packer` interface that uses this set of methods is the template `SimplePackerType`. The templates are used to generate Java classes for all primitive datatypes and objects. The second template class that is an implementation of the `Packer` interface is called `GatherPackerType` that uses `gather()` and `scatter()` methods of `mpjbuf.Buffer` class. The last template class is `MultiStridedPackerType` that uses the third set of methods provided by the buffering layer namely `strGather()` and `strScatter()`. Other implementations of the Packer interface are `NullPacker` and `GenericPacker`. The classes like `ContiguousPacker`, `IndexedPacker`, `StructPacker`, and `VectorPacker` in turn implement the abstract `GenericPacker` class.

Figure 3.3 shows the main packing and unpacking methods provided by the `Packer` interface.

The `Packer` interface is used by the sending and receiving methods to pack and unpack the messages. Consider the example of sending a message consisting of an array of integers. In this case, the datatype argument used for the standard a `Send()` method is `MPI.INT` that is an instance of `BasicType` class. A handle to a related `Packer` object can be obtained by calling the method `getPacker()`. The object `MPI.INT` is also used to get a reference to `mpjbuf.Buffer` instance by invoking the method `createWriteBuffer()`. Later a variant of the `pack()` method, shown in Figure 3.3, is used to pack the message onto a buffer that is used for communication by the underlying communication devices.

Similarly, when receiving a message with `Recv()`, the datatype object like `MPI.INT` is used to get the reference to an associated `Packer` object. A variant of the `unpack()` method is used unpack the message from `mpjbuf.Buffer` onto the user specified array.

The *contiguous datatype* consists of elements that are of same the type and at contiguous locations. Figure 3.4 shows a contiguous datatype with four elements. Each element of this datatype consists of an array of five elements. Although each element is shown as a row in a matrix, physically the datatype will be stored at contiguous locations such as an array of byte buffer.

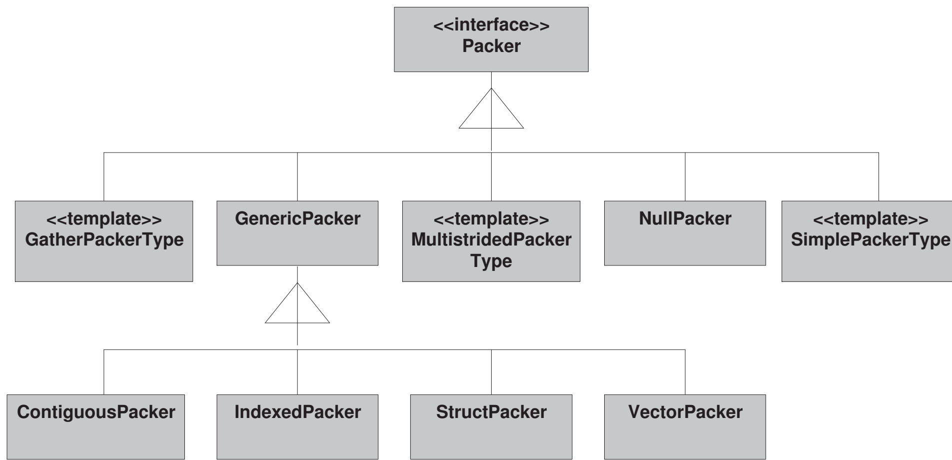

Fig. 3.2. *The Packer Class Hierarchy in MPJ Express*

```
public interface Packer {

  .. .. ..

  public abstract void pack(mpjbuf.Buffer mpjbuf, Object msg, int offset)
  public abstract void pack(mpjbuf.Buffer mpjbuf, Object msg, int offset,
                            int count)

  public abstract void unpack(mpjbuf.Buffer mpjbuf, Object msg, int offset)
  public abstract void unpack(mpjbuf.Buffer mpjbuf, Object msg, int offset,
                              int count)

  public abstract void unpack(mpjbuf.Buffer mpjbuf, Object msg, int offset)
  public abstract void unpack(mpjbuf.Buffer mpjbuf, Object msg, int offset,
                              int count)

  public abstract void unpackPartial(mpjbuf.Buffer mpjbuf, int length,
                                     Object msg, int offset)

  .. .. ..

}
```

Fig. 3.3. *The Packer Interface*

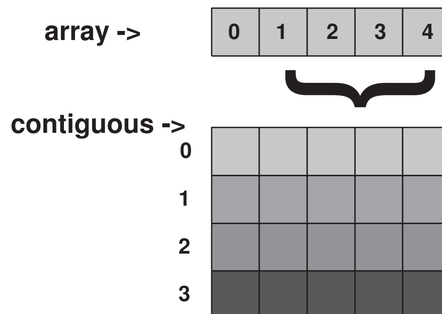

Fig. 3.4. *Forming a Contiguous Datatype Object*

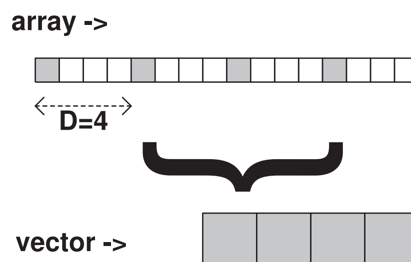

Fig. 3.5. *Forming a Vector Datatype Object*

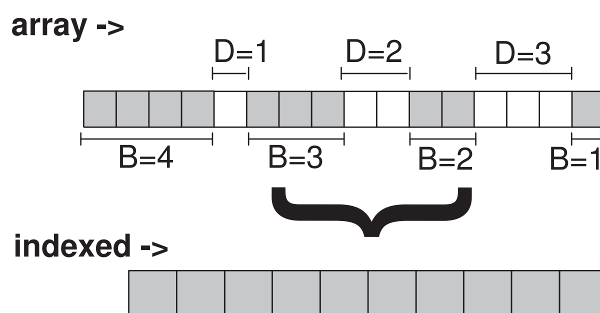

Fig. 3.6. *Forming an Indexed Datatype Object*

The *vector datatype* consists of elements that are of the same type and are found at non-contiguous locations. Figure 3.5 shows how to build an element of vector datatype from an array of primitive datatype with `blockLength=1` and `stride=4` (labelled `D` in the figure). The data is copied onto a contiguous section of memory before the actual transfer.

A more general datatype is *indexed* that allows specifying multiple block lengths and strides (also called displacement). An example is shown in Figure 3.6 with increasing displacement (starting from 0 and labelled `D`) and decreasing block length (starting from 4 and labelled `B`). The most general datatype is *struct* that not only allows varying block lengths and strides but also different basic datatypes, unlike the indexed datatype.

4. Performance Evaluation. In this section we first evaluate the performance of our buffering layer focusing on the allocation time. This is followed by a comparison of MPJ Express using combinations of direct and indirect byte buffers with our pooling strategies to find out which technique provides the best performance. We calculate transfer time and throughtput for increasing message sizes to evaluate buffering techniques. Towards the end of the section, we evaluate the performance of MPJ Express against MPICH-MX and mpiJava on Myrinet. Again, this test requires the calculation of transfer time and throughput for different message sizes.

The transfer time and throughput is calculated using a modified ping-pong benchmark. While using conventional ping-pong benchmarks, we noticed variability in timing measurements. The reason is that the network card drivers used on our cluster have a higher network latency—64 μs. The network latency of the card drivers is an attribute that determines the polling interval for checking new messages. In our modified technique, we introduced random delays before the receiver sends the message back to the sender. Using this approach, we were able to negate the effect of network card latency.

The test environment for collecting the performance results was a cluster at the University of Portsmouth consisting of 8 dual Intel Xeon 2.8 GHz PCs using the Intel E7501 chipset. The PCs were equipped with 2 Gigabytes of ECC RAM with 533 MHz Front Side Bus (FSB). The motherboard (SuperMicro X5DPR-iG2) was

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 2.4.32 Linux kernel. The software used for the Intel Gigabit adaptor was the proprietary Intel e-1000 device driver. The JDK version used for tests on mpiJava and MPJ Express was Sun JDK 1.5 (Update 6). The C compiler used was GNU GCC 3.3.5.

4.1. Buffering Layers Performance Evaluation. In this section, we compare the performance of our two buffering strategies with direct allocation of `ByteBuffer`s. We are also interested in exploring the performance difference between using direct and indirect byte buffers in MPJ Express communication methods. There are six combinations of our buffering strategies that will be compared in our first test—Buddy1, Buddy2, and a simple allocation scheme, each using direct and indirect byte buffers.

4.1.1. Simple Allocation Scheme Time Comparison. In our first test, we compare isolated buffer allocation times 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 chunk of 8 Mb before the next allocation occurs), or the reference is freed in the case of straightforward `ByteBuffer` allocation.

Figure 4.1 shows a comparison of allocation times. It should first be noted that all the buddy-based schemes are dramatically better than relying on the JVMs 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 are included in the results in an averaged sense; this is a fair representation of what will happen in a real application. Generally we attribute the dramatic increase in average allocation time for large `ByteBuffer`s to forcing proportionately many garbage collection cycles. All the buddy variants (by design) avoid this overhead. The allocation times for buddy based schemes decrease for larger buffer sizes because less time is spent in traversing the data structures to find an appropriately sized buffer. The size of the initial region is 8 Mb—resulting in the least allocation time for this buffer size. The best strategy in almost all cases is Buddy1 using direct buffers.

Quantitative measurements of the memory footprint suggest the current implementation of Buddy2 also has about a 20% larger footprint because of the extra objects stored. In its current state of development, Buddy2 is clearly outperformed by Buddy1. But there are good reasons to believe that with further development, a variant of Buddy2 could be faster than Buddy1. This will be the subject of future work.

4.1.2. Incorporating Buffering Strategies into MPJ Express. 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. The reason for performing this test is to see if there are any performance benefits of 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 it to or reading it from the socket. Thus, we expect to see an overhead of this additional copying for indirect buffers.

Figure 4.2 shows transfer time comparison on Fast Ethernet with different combinations of buffering in MPJ Express. Normally transfer time comparison is useful for evaluating the performance on smaller messages. We do not see any significant performance difference for small messages.

Figure 4.3 shows the throughput comparison. Here, MPJ Express 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 128 Kb message size is because of the change in communication protocol from eager send to rendezvous.

4.2. Evaluating MPJ Express using Myrinet. In this test we evaluate the performance of MPJ Express against MPICH-MX and mpiJava by calculating the transfer time and throughput. We used MPJ Express (version 0.24), MPICH-MX (version 1.2.6..0.94), and mpiJava (version 1.2.5) on Myrinet. We also added *mpjdev* [13] to our comparison to better understand the performance of MPJ Express. MPJ Express uses

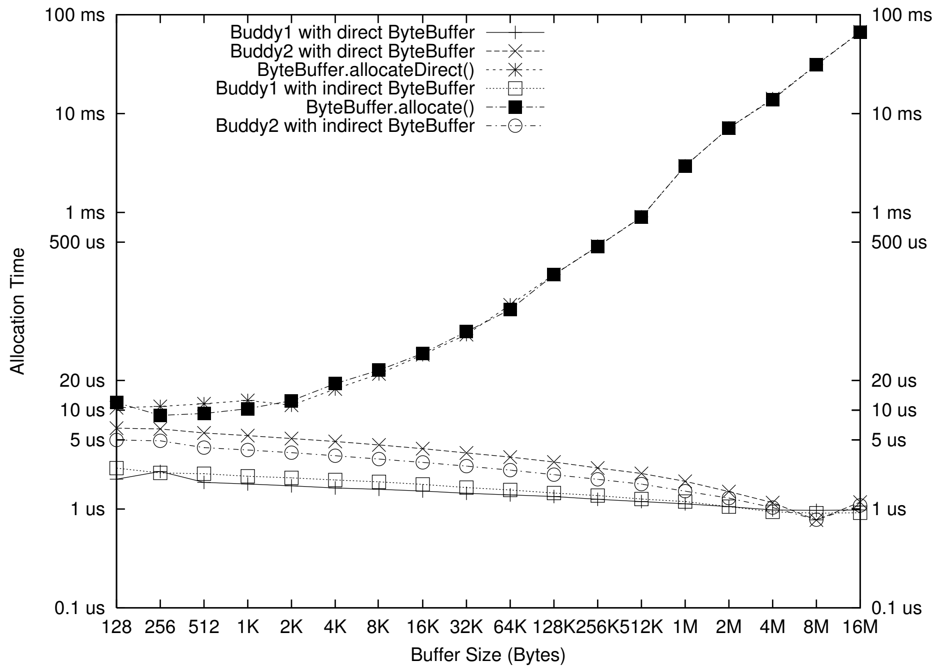

FIG. 4.1. *Buffer Allocation Time Comparison*

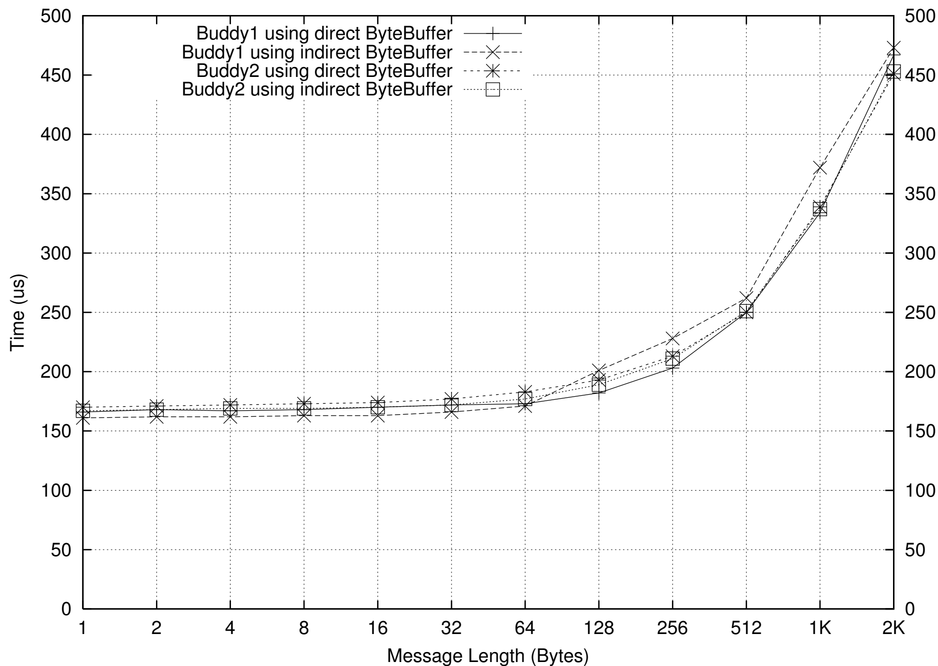

FIG. 4.2. *Transfer Time Comparison on Fast Ethernet*

mpjdev, which in turn relies on *mxdev* on Myrinet. These tests were conducted on the same cluster using the 2 Gigabit Myrinet eXpress (MX) library [17] version 1.1.0.

Figure 4.4 and Figure 4.5 show the transfer time and throughput comparison. The latency of MPICH-MX is 4 μs. MPJ Express and mpiJava have a latency of 23 μs and 12 μs, respectively. The maximum throughput of MPICH-MX was 1800 Mbps with 16 Mbyte messages. MPJ Express achieves a maximum of 1097 Mbps for the same message size. mpiJava achieves a maximum of 1347 Mbps for 64 Kbyte messages. After this,

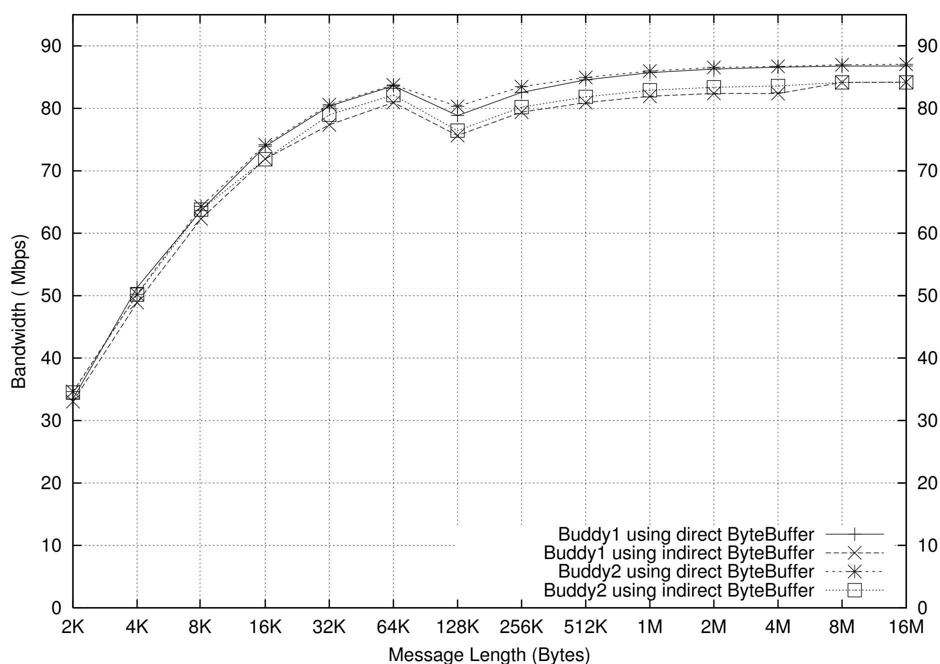

Fig. 4.3. *Throughput Comparison on Fast Ethernet*

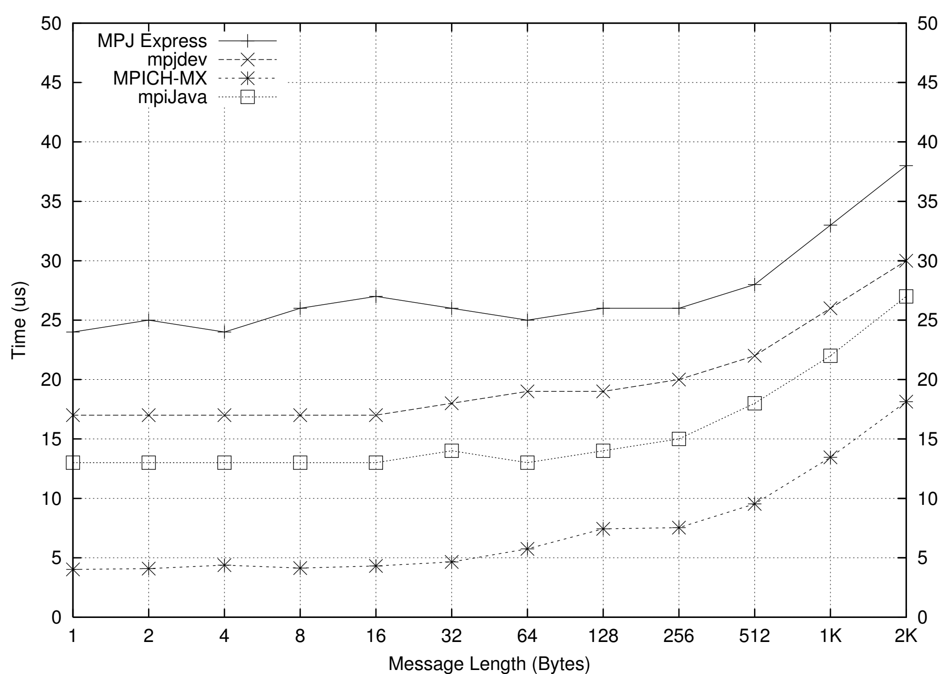

Fig. 4.4. *Transfer Time Comparison on Myrinet*

there is a drop, bringing throughput down to 868 Mbps at 16 Mbyte message. Throughput starts decreasing as the message size increases from 64 Kbytes. This is primarily due to copying data between the JVM and OS. Although we are using JNI in the MPJ Express Myrinet device, we have been able to avoid this overhead by using direct byte buffers. However, other overheads of JNI, such as the increased calling time of methods are visible in the results. The mpjdev device, that sits on top of mxdev, attains a maximum throughput of 1826 Mbps for 16 Mbyte messages, which is more than that of MPICH-MX. Our device layer is able to make the most

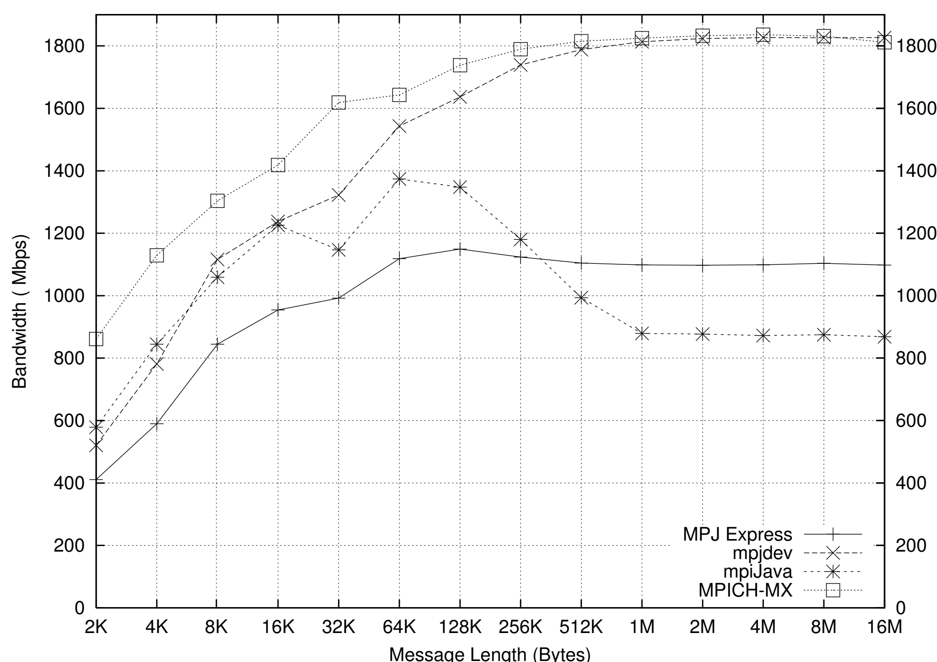

Fig. 4.5. *Throughput Comparison on Myrinet*

of Myrinet. This shows the usefulness of our buffering API, because the message has already been copied onto a direct byte buffer. It is clear that a combination of using direct byte buffers and JNI incurs virtually no overhead.

The difference in the performance of MPJ Express and mpjdev shows the packing and unpacking overhead incurred by the buffering layer. Besides this overhead, this buffering layer helps MPJ Express to avoid the main data copying overhead of JNI—MPJ Express achieves a greater bandwidth than mpiJava. Secondly, such a buffering layer is necessary to provide communication of derived datatypes. A possible fix for this overhead is to extend mpiJava 1.2 and the MPJ API to support communication to and from `ByteBuffer`s.

5. Conclusions and Future Work. MPJ Express is our implementation of MPI-like bindings for the Java language. As part of this system, we have implemented a buffering layer that exploits direct byte buffers for efficient communication on proprietary networks, such as Myrinet. Using this kind of buffer has enabled us to avoid the overheads of JNI. In addition, our buffering layer helps to implement derived datatypes in MPJ Express. Arguably, communicating Java objects can achieve the same effect as communicating derived types, but we have concerns related to notoriously slow Java object serialization.

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 an application level memory management technique instead of relying on the JVM'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 have evaluated the performance of MPJ Express against other messaging systems. MPICH-MX achieves the best performance followed by MPJ Express and mpiJava. By noting the difference between MPJ Express and the mpjdev layer, we have identified a certain degree of overhead caused by additional copying in our buffering layer. We aim to resolve this problem by introducing methods that communicate data to and from `ByteBuffer`s.

We released a beta version of our software in early September 2005. The current version provides communication functionality based on a thread-safe Java NIO device.The current release also contains our buffering API with the two implementations of buddy allocation scheme. This API is self-contained and can be used by other Java applications for application level explicit memory management.

MPJ Express can be downloaded from `http://mpj-express.org`.

REFERENCES

[1] M. Aldinucci, M. Danelutto, and P. Teti, *An advanced environment supporting structured parallel programming in Java*, Future Generation Computer Systems, 19 (2003), pp. 611–626.
[2] M. Alt and S. Gorlatch, *A prototype grid system using Java and RMI*, in Parallel Computing Technologies (PaCT 2003), vol. 2763 of Lecture Notes in Computer Science, Springer, 2003, pp. 401–414.
[3] M. Baker, B. Carpenter, G. Fox, S. H. Ko, and S. Lim, *An Object-Oriented Java interface to MPI*, in International Workshop on Java for Parallel and Distributed Computing, San Juan, Puerto Rico, April 1999.
[4] B. Blount and S. Chatterjee, *An Evaluation of Java for Numerical Computing*, in ISCOPE, 1998, pp. 35–46.
[5] O. Bonorden, J. Gehweiler, and F. M. auf der Heide, *A Web computing environment for parallel algorithms in Java*, Scalable Computing: Practice and Experience, 7 (2006).
[6] B. Carpenter, G. Fox, S.-H. Ko, and S. Lim, *mpiJava 1.2: API Specification*.
[7] C.-C. Chang and T. von Eicken, *Javia: A Java Interface to the Virtual Interface Architecture*, Concurrency—Practice and Experience, 12 (2000), pp. 573–593.
[8] M. Danelutto and P. Dazzi, *Joint structured/unstructured parallelism exploitation in muskel*, in Third International Workshop on Practical Aspects of High-Level Parallel Programming (PAPP 2006), V. N. Alexandrov et al., eds., vol. 3992 of Lecture Notes in Computer Science, Springer, 2006, pp. 937–944.
[9] R. Gordon, *Essential JNI: Java Native Interface*, Prentice Hall PTR, Upper Saddle River, NJ 07458, 1998.
[10] Y. Gu, B.-S. Lee, and W. Cai, *JBSP: A BSP programming library in Java*, Journal of Parallel and Distributed Computing, 61 (2001), pp. 1126–1142.
[11] R. Hitchens, *Java NIO*, O'Reilly & Associates, 2002.
[12] D. Knuth, *The Art of Computer Programming: Fundamental Algorithms*, Addison Wesley, Reading, Massachusetts, USA, 1973.
[13] S. Lim, B. Carpenter, G. Fox, and H.-K. Lee, *A Device Level Communication Library for the HPJava Programming Language*, in IASTED International Conference on Parallel and Distributed Computing and Systems (PDCS 2003), November 2003.
[14] Mark Baker, Bryan Carpenter, Aamir Shafi, *MPJ Express: Towards Thread Safe Java HPC*, in Proceedings of the 2006 IEEE International Conference on Cluster Computing (Cluster 2006), IEEE Computer Society, September 2006, pp. 1–10.
[15] Message Passing Interface Forum, *MPI: A Message-Passing Interface Standard*, University of Tenessee, Knoxville, TN, June 1995. `http://www.mcs.anl.gov/mpi`
[16] *Myricom*. http://www.myri.com.
[17] *The MX (Myrinet eXpress) library*. `http://www.myri.com/scs/MX/mx.pdf`
[18] M. Welsh and D. Culler, *Jaguar: Enabling Efficient Communication and I/O in Java*, Concurrency: Practice and Experience, 12 (2000), pp. 519–538.

Edited by: Anne Benoît and Frédéric Loulergue
Received: September, 2006
Accepted: March, 2007