An Approach to Buffer Management in Java HPC Messaging

Mark Baker, Bryan Carpenter and 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 lower 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 API. We evaluate the performance of our buffering layer and demonstrate the usefulness of direct byte buffers.

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 mostly adopted conventional (sequential) languages, with programs developed in relatively conventional programming environments usually supplemented by libraries like MPI that support parallel programming. 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. Also, it supports multi-threading and provides simple primitives like `wait()` and `notify()` that can be used to synchronize access to shared resources. Recent JDKs (Java Development Kits) 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.

We have developed MPJ Express (MPJE) [6], a thread-safe implementation of Java MPI API. A challenging aspect of implementing Java HPC messaging software is providing an efficient intermediate buffering layer. 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 [7], 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. Until quite recently doing this kind of thing in Java was hard.

JDK 1.4 introduced Java New I/O (NIO) [8]. In NIO, read and write methods on files and sockets (for example) are mediated through a family of buffer classes handled specially 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` write method. 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 [9] incurred by additional copying when using Java Native Interface (JNI) [4]. On the other hand, it may be preferable to create a native buffer using the JNI. These native buffers can be useful for a native MPI or a proprietary network device.

We are convinced that NIO provides essential ingredients [2] of an efficient messaging system via non-blocking I/O and direct buffers.

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 MPJE 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. Our buffering strategy uses a pooling mechanism to avoid creating a buffer instance 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 [5], 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. In addition, 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.

The remainder of 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 [1], 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 may be required between the JVM and the native C library. 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 [9]. 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.

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.

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 dynamic buffer is used to store serialized Java objects, where it is not possible to determine the length of the serialized objects beforehand. The class structure of our package is shown in Figure 1.

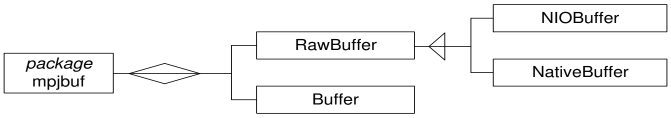

Fig. 1. Primary Buffering Classes in `mpjbuf`

3.1 Memory Management

We have implemented our own application level memory management mechanism based on a buddy allocation scheme [5]. 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`.

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 `NIOBuffer`.

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

In a 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.

The First Pooling Strategy Our first implementation (called Buddy1 below) 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.

Figure 2 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

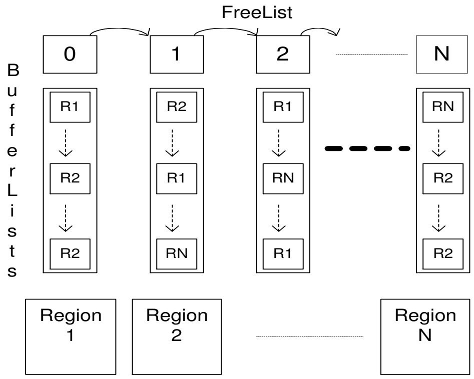

Fig. 2. The First Implementation of Buffer Pooling

with `FreeList`. An initial chunk of memory of size M is allocated. At this point, `BufferList`s 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 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 `BufferList`s are independent of a region and these lists grow as new regions are created to match user requests.

The Second Pooling Strategy Our second implementation (called Buddy2 below) stores higher-level buffer abstractions (`NIOBuffer`) in `BufferList`s. Unlike the first strategy, each region has its own `FreeList` and has a pointer to the next region as shown in Figure 3. 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.

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, Buddy2, and simple-minded allocation all using direct and indirect byte buffers.

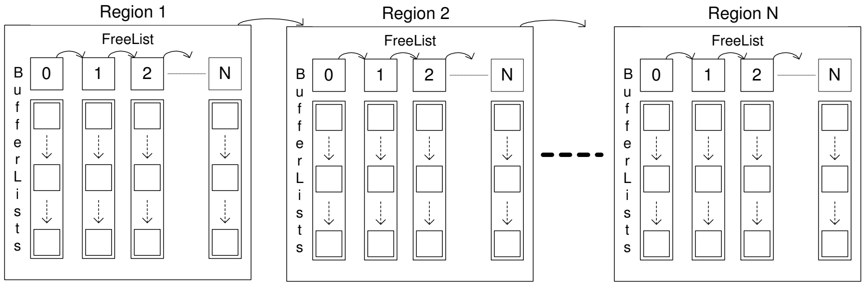

Fig. 3. The Second Implementation of Buffer Pooling

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 Mbytes before the next allocation occurs), or the reference is freed in the case of straightforward `ByteBuffer` allocation.

Figure 4 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 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 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 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 Mbytes—resulting in the least allocation time for this buffer size. 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.

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 is future work.

4.2 Incorporating Buffering Strategies into MPJE

In this test, we compare throughput measured by a simple ping-pong benchmark using each of the different buffering strategies. These tests were performed on

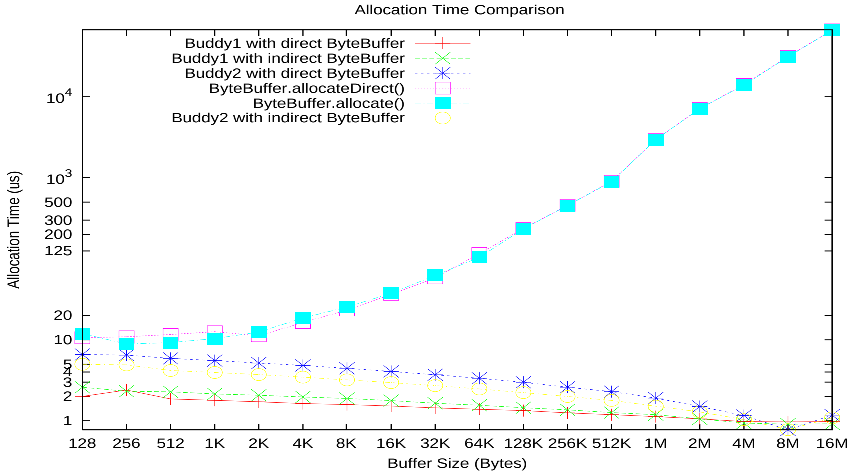

Fig. 4. Allocation Time Comparison

Fast Ethernet. 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 ByteBuffers 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.

Figure 5 shows that 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 an application level memory management technique instead of relying on 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.

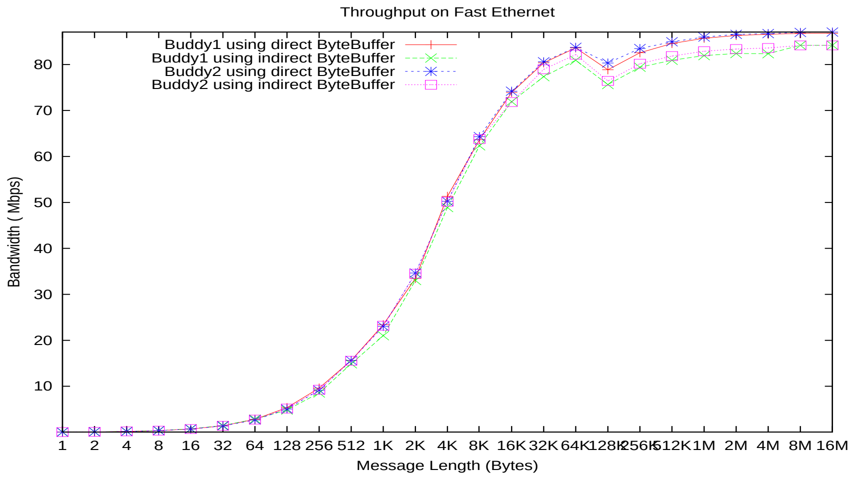

Fig. 5. Throughput Comparison

We released a beta version of our software in early September 2005. This release 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 memory management. Currently, we are working to release additional messaging devices based on mpiJava and Myrinet eXpress(MX).

References

1. Mark Baker, Bryan Carpenter, Geoffrey Fox, Sung Hoon Ko, and Sang Lim. An object-oriented Java interface to MPI. In *International Workshop on Java for Parallel and Distributed Computing*, San Juan, Puerto Rico, April 1999.
2. Mark Baker, Hong Ong, and Aamir Shafi. A status report: Early experiences with the implementation of a message passing system using Java NIO. Technical Report DSGTR06102004, DSG, October 2004. http://dsg.port.ac.uk/projects/mpj/docs/res/DSGTR06102004.pdf.
3. Chi-Chao Chang and Thorsten von Eicken. Javia: A Java interface to the virtual interface architecture. *Concurrency - Practice and Experience*, 12(7):573–593, 2000.
4. The Java Native Interface Specifications. http://java.sun.com/j2se/1.3/docs/guide/jni.
5. Donald Knuth. *The Art of Computer Programming: Fundamental Algorithms*. Addison Wesley, Reading, Massachusetts, USA, 1973.
6. MPJ Express. http://dsg.port.ac.uk/projects/mpj.
7. Myricom, The MX (Myrinet eXpress) library. http://www.myri.com.
8. The Java New I/O Specifications. http://java.sun.com/j2se/1.4.2/docs/guide/nio.
9. Matt Welsh and David Culler. Jaguar: enabling efficient communication and I/O in Java. *Concurrency: Practice and Experience*, 12(7):519–538, 2000.