

vrije Universiteit *amsterdam*



Using the new Java™ I/O interface in parallel computing

Niels Drost

Abstract

Recently, the Java™ programming language was extended with new I/O functionality, dubbed NIO, especially designed for speed and scalability. This thesis explores the different possible uses for this new functionality in parallel computing. As a test case we integrate NIO in Ibis, a programming environment especially designed for parallel and grid applications. We take two approaches in this integration. First, we use NIO to optimize conversion of data to and from bytes in an existing Ibis implementation. Secondly, we create a completely novel Ibis implementation, using NIO to implement all communication, including multicast and many to one style functionality. We show that NIO has considerable advantages and improves both throughput and scalability.

Contents

1	Introduction	1
2	The new I/O package in Java™ 1.4	3
2.1	Buffers	3
2.2	Channels	3
2.3	Selecting	4
2.4	Views	4
2.5	Comparison of java.nio with java.io	5
3	Ibis: a Grid Programming Environment	7
3.1	The Ibis Portability Layer	8
3.2	Implementations	8
4	Using NIO in existing Ibis implementations	11
5	NioIbis: a new ibis implementation	17
5.1	Serialization	17
5.2	The Send/Receive Thread	18
5.3	Thread Overview	20
5.4	NioIbis Send Ports	21
5.5	NioIbis Receive Ports	22
6	Experimental Results	25
6.1	Conversion	25
6.2	Serialization	26
6.3	Throughput and Latency	27
6.4	A Master/Worker benchmark	29
6.5	Successive Overrelaxation	30
6.6	Barnes-Hut	31
7	Conclusions and Future Work	33
8	Acknowledgments	35
	Bibliography	36

Chapter 1

Introduction

In recent years, the Java™ programming language platform evolved more and more into a general use programming environment. After a, literally, slow start, virtual machine performance has improved significantly. This resulted in Java programs becoming as fast as comparable programs written in C.

Although Java’s raw processing speed of statements was on a par with other programming languages, communication and I/O performance was not. The overhead of communicating with another computer using standard *Java Remote Method Invocation*(RMI) was orders of magnitude higher than solutions written in other languages. This was partly because the Sun RMI implementation was far from optimized for high performance computing, and partly because of limitations in the I/O model as used by Java. Work has been done to optimize the implementation of RMI [5], but the I/O model limitations remained.

To address the problems associated with the *java.io* package, a Java Specification Request(JSR) was established using the Java Community Process. This JSR-51¹, called for

“APIs for scalable I/O, fast buffered binary and character I/O, regular expressions, charset conversion, and an improved filesystem interface.”

A subset of JSR-51 was implemented in Java 1.4, mostly in the new *java.nio* package. Although usually referred to as *New I/O*(NIO), it is not meant to completely replace the existing I/O mechanisms. Instead, it tries to complement the existing I/O package as much as possible. Chapter 2 gives an overview of NIO.

The goal of the thesis is to investigate if it is possible to have faster, more scalable communication in parallel computing using the new communication methods made available in the Java language by NIO. By faster we mean having a higher throughput and a lower latency when transferring data between two computers. By more scalable we mean handling more connections and data transfers to and from a single computer than currently possible.

Our goal implies we will mostly make use of the *scalable I/O* part of JSR-51. The rest of the JSR, although useful for many applications, does not help to achieve these goals. One distinct feature of NIO we are interested in is the

¹<http://jcp.org/en/jsr/detail?id=51>

select mechanism. This mechanism allows a single thread to handle multiple connections, something not possible with the *java.io* package.

The platform on which we will perform our research is the *Ibis* [9] programming environment. Chapter 3 serves as a short introduction to *Ibis*. An overview of the modifications and additions made to the *Ibis* environment as a result of our research is given in Chapters 4 and 5, followed by the results of running a number of benchmarks and programs to evaluate the resulting system in Chapter 6. Finally, in Chapter 7 conclusions are drawn from these results and recommendations for future work in this area are given.

Chapter 2

The new I/O package in Java™ 1.4

This chapter describes the principles behind and elements of the *java.nio* package in Java™1.4. For more information, see the entire specification [8] and the book by Ron Hitchens [3]

2.1 Buffers

The central notion in NIO is the *buffer*. The *Buffer* class is a simple data storage interface, designed to hide the actual underlying storage, called the *backing store*, from the user. The interface has *put* and *get* functions for storing and retrieving data. It has the notion of a *position* to keep track of where to put data, and a *limit* to denote the first position in the buffer which should not be used. The limit may be less than the actual size, or *capacity*, of a buffer.

There are various implementations of the buffer interface, for example, a simple wrapper around an array and a memory mapped file. Using some simple Java Native Interface(JNI) calls, it is possible to use an arbitrary piece of memory as a buffer, for example, video memory.

When creating a buffer of the simplest type, which is backed by memory, a distinction is made between *direct* and *non-direct* buffers. Non-direct buffers use standard virtual machine memory, similar to any other object in Java. Direct buffers are also able to use other memory which is normally not usable by Java programs. For example, a virtual machine implementation can use memory *pinned* by the operating system. This memory will never be moved or swapped out, making it possible to directly perform I/O operations, like DMA, on this memory. This potentially makes operations on the buffer much more efficient.

2.2 Channels

Channels are used to send and receive data stored in buffers. The *Channel* interface is a general interface, with *write* and *read* functions to transfer data. How this data is then transferred depends on the actual channel type. Several channel implementations exist. One sends data over a TCP connection. Another

uses UDP. A third implements a *pipe* channel which is useful for communicating within a virtual machine. A channel which transfers data to and from files is also provided.

The *write* and *read* functions of the channel interface are able to transfer an entire buffer or parts of it. In general, they start transferring data at the *position* of a buffer, and stop when they reach the *limit*. The position is updated after the transfer to reflect the data transferred. If a buffer is given to a channel with a position that is equal to the limit, it will simply return without transferring any data at all.

One feature of channels which is useful for sending a large amount of data is the *scatter/gather* interface. Using a special version of the *write* and *read* functions it is possible to send out or receive an array of buffers instead of the usual single buffer. Since most modern operating systems support this type of communication natively, this improves efficiency greatly over sending one buffer at a time.

2.3 Selecting

Some types of channel support *non-blocking* as well as *blocking* communication. When non-blocking communication is used a channel will always directly return from a read or write operation, even if this means transferring no data at all. Blocking communication, as used by Java traditionally, always blocks until at least some data is transferred.

To make sure a *write* or *read* actually transfers data when in non-blocking mode, a mechanism for *selecting* is provided (similar to the POSIX notion). A select operation can be performed on one or more channels. For each of these channels one or more operations can be included in the select operation. The following operations are defined:

read from a channel

write to a channel

connect to a peer channel

accept a new connection from a peer

To use the select mechanism, channels must first be registered with a *selector*, and it must be specified which operations are of interest for each channel. A *select* function is then available to generate a list of all the channels with operations ready. This function can be instructed to block forever, for a limited time, or not at all while waiting for one or more channels to become ready. The resulting *ready set* contains every channel which is able to do one or more operations without blocking. It also specifies exactly which operations are ready for each channel.

2.4 Views

Buffers are typed, i.e., they can only contain elements of a single type. Buffer implementations exist for all primitive types, except *boolean*, which wasn't deemed

useful. Most interaction with channels, as well as the creation of direct buffers, can only be done with buffers of the *byte* type. To use other buffer types with a channel a *view* can be created of a byte buffer. This view can be of any buffer type, using the same backing store as the buffer it was created from. The new buffer's position, limit and such are independent of the old buffer.

For example, views make it possible to create a view of a byte buffer which contains floating point numbers or *floats*. As the byte buffer uses the same memory for storage, floats put into the float buffer will appear in the byte buffer converted to bytes. No explicit conversion of the data is necessary.

Another, related, way to make a view is to actually copy, or *duplicate* a buffer. The new buffer is an exact copy of the duplicated buffer, and uses the same backing store. It is even possible to *slice* a buffer, so that the newly created buffer comprises of a section of the original buffer. It too shares its memory with the buffer it was created from.

2.5 Comparison of java.nio with java.io

This section describes a simple throughput benchmark. It is meant to test some capabilities of NIO, and give a general idea about the speed of NIO. We compare it to a similar program written using the *java.io* package.

Although written using different types of I/O, both benchmarks have an identical structure. Using two programs (which may be on the same machine) they transfer an array of doubles across a TCP connection. They do this by first converting the data to bytes, sending it to the destination, and converting the received data to doubles again.

The *java.io* version of the benchmark does conversion by first converting the double to a long using the *doubleToLongBits* function found in the *Double* class. Using mostly logical functions such as *shift* and *or* the bytes are then extracted from the long.

The NIO version of the benchmark uses views to convert the data. It creates a byte buffer and a double view of that buffer. To send out the data, it puts it in the double buffer. The data is then automatically converted to bytes and made available in the byte buffer. It then writes the byte buffer through the channel to the network.

The sender and receiver of the benchmark were run on the same dual Pentium III computer, while using the loopback device to communicate. The *java.io* version of the benchmark achieved a throughput of 11.9 Mbyte/s while the NIO version transferred data at 53.3 Mbyte/s. This shows NIO has a great advantage over using *java.io*, at least when used for this type of communication. Since the TCP implementation of NIO shares the code to actually sent out data with *java.io*, this difference is entirely caused by the increased conversion performance of NIO.

Chapter 3

Ibis: a Grid Programming Environment

As a platform to test the capabilities of NIO we will be using the *Ibis* [9] environment. Ibis is a grid programming environment specifically designed to be both flexible and efficient. Usually, systems written for a grid are optimized either for portability or performance, but not both. Pure Java implementations of Ibis are available. Programs written for the Ibis platform are therefore able to run “everywhere”. Ibis simultaneously provides implementations which use highly optimized but non-standard solutions for increased performance in special cases. This approach makes Ibis a both portable and efficient platform for grid computing.

Several programming models are implemented on top of Ibis. Examples are Java RMI and GMI [4], an RMI-like interface that has support for group communication. Another model is an implementation of the *Satin*[10] divide-and-conqueror system. In this system, communication between the different nodes in the system is completely hidden from the user.

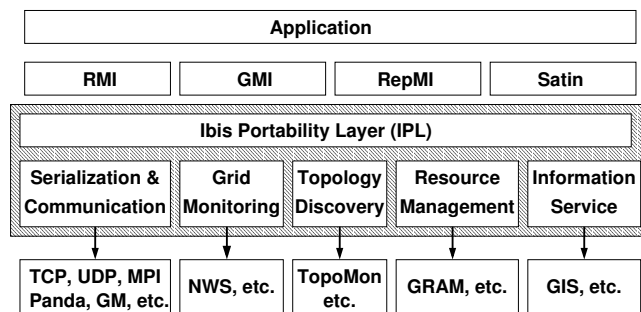


Figure 3.1: Design of Ibis.

3.1 The Ibis Portability Layer

One of the central notions in Ibis is the *Ibis Portability Layer (IPL)*. Figure 3.1 shows the relation of the IPL to the rest of Ibis. It acts as an interface between the different run time systems on top of it, and the implementations for different architectures below. This makes it possible to do the selection of a implementation at run time, based on the needs of the run time system, and the specific system Ibis is run on. For instance, a local high speed network such as Myrinet [2] might be available.

Communication primitives are provided by the IPL in the form of send ports and receive ports. These ports can be connected to each other to form *unidirectional message channels*. A new (empty) message can be requested from a send port. This message can then be filled with data. All primitive data types in Java are supported, as well as objects. When all data is put into the message it can be sent. All the data in the message will be serialized to some transferable format by the send port and sent over the network to the receive port.

Receiving of a message at the receive port can be done in two ways. One is to explicitly receive a message. It is possible to specify a timeout on the receiving of a message. The other method for receiving a message is to use an *upcall handler*. The receive port will initiate an upcall whenever a message is received. It is guaranteed only one message is alive at any time. When a message is received its data is de-serialized and made available for reading by the receiver.

A send port can be connected to multiple receive ports and a receive port is able to have multiple incoming connections from send ports. This makes it possible to support multicast and many-to-one communication in a very direct and efficient manner.

Send and receive ports are typed. Only send and receive ports of the same type can be connected to each other. A *PortType* can be configured using *properties*. Configurable properties include reliability, ordering of messages, serialization type and multicast support. These properties are used by the Ibis implementation to select a suitable implementation for a send or receive port when one is created.

3.2 Implementations

This section gives a short overview of the various Ibis implementations. Most Ibis implementations make use of a standard serialization method provided by the Ibis framework. This *Ibis Serialization* converts the primitive data and objects given to it to a number of primitive arrays. These arrays can then be further processed by the implementation to a sendable form. Ibis is also able to use the default serialization incorporated in Java. This is mostly used for comparison with Ibis serialization, but is also useful when versioning is needed, which is currently not supported by Ibis serialization.

TcpIbis

TcpIbis is the most general Ibis implementation. As it only uses standard Java code, and communicates via TCP, it is possible to use this Ibis implementation

on virtually any computer system, as long as a Java virtual machine is available. Ibis Serialization, which is used by TcpIbis, outputs data in the form of arrays of primitive types. TCP Sockets, however, are only able to send and receive bytes. The TcpIbis implementation therefore *converts* the data given to it by Ibis Serialization into bytes.

Since multicast is not supported by the TCP Sockets model, one-to-many communication is not supported natively by TcpIbis. When a send port is connected to more than one receive port, messages are sent out to each receive port separately. This makes multicast quite expensive in TcpIbis.

The TcpIbis receive port uses blocking communication to receive data. To support multiple incoming connections TcpIbis uses a thread per connection which waits for incoming messages from that connection. When a new message is received, it is given to the receive port using a synchronized method.

MessagePassingIbis

The Ibis message passing implementation acts as a general implementation for transferring data using message passing libraries. It is currently able to use either Panda [1] or MPI [6] to transfer data.

As the message passing libraries supported by this implementation are written in C, conversion is not as big of a problem as it is in a full Java implementation. Simple pointer manipulation is usually sufficient to create a byte view of a data structure.

Multicast is also handled by most libraries natively, so no repeated sending of messages is needed. If many-to-one communication is needed, some thread switching might be required to enable the right receive port to process a received message.

NetIbis

NetIbis is an Ibis implementation designed to use multiple network types and protocols simultaneously. This is useful in a heterogeneous system. It currently supports GM (a Myrinet communication library), TCP and UDP. NetIbis uses a *driver* model. Each of these drivers implements a specific protocol. A special type of drivers are *virtual drivers*. These do not actually transfer data to and from a network, but add some functionality to an existing driver. For example, a virtual driver may implement reliability, or multicast. By stacking one or more drivers on top of each other it is possible to get any functionality needed by the user. Serialization is implemented in NetIbis as a virtual driver.

Chapter 4

Using NIO in existing Ibis implementations

In the following two chapters we will explore the possibilities of using NIO in the Ibis project. We will use two approaches. In this chapter we will explore the possibilities of adding features of NIO to existing Ibis implementations. In the next we will describe a completely new Ibis implementation.

One approach to using NIO in an existing Java program without having to rewrite most of the program is to use NIO only for *conversion* of primitive types to and from bytes. This functionality can be added with little modification to the existing implementation. Based on the results of the benchmark in Section 2.5 it should improve the performance of the implementation by a considerable amount.

In *TcpIbis*, the TCP based Ibis implementation, all conversion is done using the *Conversion* class. We modified this class to make use of NIO. The class is now an abstract class which defines the interface used for conversion. Multiple implementations of this abstract class exist. One implementation, called *simple conversion*, does conversion using the original conversion code. Two others use NIO buffers and views. The fourth and last is a hybrid, using simple conversion for small amounts of data, and NIO for large. Of all implementations two versions exist, one for big endian and one for little endian byte order. Generally, conversion to the native byte order of a platform is much faster than converting to the opposite byte order. When conversion needs to be done to a non native byte order, all the bytes must be "swapped", causing a degradation of performance.

To clarify the implementations, we will discuss the code needed to convert an array of integers to an array of bytes in the various implementations. The first implementation, shown in figure 4.1, does not use NIO. It is relatively straightforward. For each element of the array that needs to be converted, each byte is calculated by shifting the integer with an appropriate amount, and masking all but the last byte.

Figure 4.2 shows the first implementation using NIO, the *NIO/Wrap* implementation. The constructor of the *NioWrapLittleConversion* class (line 11-21) allocates a buffer (line 16) to use as temporary storage space. Because it uses the *allocateDirect* function to do this, the memory allocated will be more effi-

```

public class SimpleLittleConversion extends Conversion {

    public void int2byte(int [] src, int off, int len,
                       byte [] dst, int off2) {

        int v = 0;
        int count = off2;

        for (int i=0;i<len;i++) {
            v = src[off+i];

            dst[count+0] = (byte)(0xFF & v);
            dst[count+1] = (byte)(0xFF & (v >> 8));
            dst[count+2] = (byte)(0xFF & (v >> 16));
            dst[count+3] = (byte)(0xFF & (v >> 24));
            count += INT_SIZE;
        }
    }
}

```

Figure 4.1: Excerpt of a conversion class using standard Java code

cient when doing I/O operations on it. The virtual machine may, for example, pin the memory so it will not be swapped out or moved. Allocating memory this way takes longer than allocating normal memory though. Since we only allocate it once, at the initialization of the Ibis implementation, this trade-off is acceptable here. The byte order of the buffer is set to little endian. On line 20 an integer buffer is made as a view of the byte buffer.

The actual conversion of the data is done in one of two ways depending on the size of the data being converted. The first solution (lines 26-29) is to *wrap* the destination array into a byte buffer, create an integer view from it, and put the data into the view. This method of conversion requires the data to be copied once in addition to the creation of a buffer and a view buffer. The buffer returned when wrapping an array is a non-direct buffer, making operation on this buffer less efficient. The second solution (lines 31-35) *copies* the data into the global integer buffer view and extracts the bytes from the global byte buffer. Since the position and limit of the byte buffer and its view are independent the position and limit of the byte buffer are adjusted on line 34 to reflect the data now present in the buffer before extracting the data on line 35. This second method of converting the data has the advantage of not having to allocate any buffers, but it does copy the data twice. The *wrap* solution is used for large arrays, and the *copy* solution is used for smaller ones.

The second implementation using NIO is called *NIO/Chunk* conversion and is shown in Figure 4.3. The constructor is identical to the first NIO implementation, and is not shown here. This implementation does not use *wrapping* when an array is larger than the global buffer. Instead, it divides the array into *chunks* and converts them one by one, using the global buffer for each conversion. Line 7 shows the while loop which exits when all data has been converted. It determines the chunk size at line 8, and performs the conversion of the chunk at lines 10-14. Lines 16-18 update the offsets and decrease the size of the data that still needs to be converted.

Benchmarks (see section 6.1) show that the NIO implementations are not

```

public final class NioWrapLittleConversion
    extends SimpleLittleConversion {

    public static final int BUFFER_SIZE = 8 * 1024;

    private final ByteOrder order;

    private final ByteBuffer byteBuffer;
    private final IntBuffer intBuffer;

    public NioWrapLittleConversion () {

        order = ByteOrder.LITTLE_ENDIAN;

        //allocate byte buffer, and set its byte order
        byteBuffer = ByteBuffer.allocateDirect(BUFFER_SIZE)
            .order(order);

        //create a view of the byte buffer
        intBuffer = byteBuffer.asIntBuffer();
    }

    public void int2byte(int [] src, int off, int len,
        byte [] dst, int off2) {
        if(len > (BUFFER_SIZE / INT_SIZE)) {
            //given array too large to fit into buffer
            IntBuffer buffer = ByteBuffer.wrap(dst, off2,
                len * INT_SIZE).order(order).asIntBuffer();
            buffer.put(src, off, len);
        } else {
            intBuffer.clear();
            intBuffer.put(src, off, len);

            byteBuffer.position(0).limit(len * INT_SIZE);
            byteBuffer.get(dst, off2, len * INT_SIZE);
        }
    }
}

```

Figure 4.2: NIO conversion class using wrapping for large arrays

```

public final class NioChunkLittleConversion
    extends SimpleLittleConversion {

    //global variables, constructor

    public void int2byte(int [] src, int off, int len,
        byte [] dst, int off2) {

        while(len > 0) {
            int chunkSize = Math.min(BUFFER_SIZE / INT_SIZE, len);

            intBuffer.clear();
            intBuffer.put(src, off, chunkSize);

            byteBuffer.position(0).limit(chunkSize * INT_SIZE);
            byteBuffer.get(dst, off2, chunkSize * INT_SIZE);

            len -= chunkSize;
            off += chunkSize;
            off2 += chunkSize * INT_SIZE;
        }
    }
}

```

Figure 4.3: NIO conversion class converting large arrays in chunks

```

public final class HybridChunkLittleConversion
    extends SimpleLittleConversion {

    public static final int THRESHOLD = 512; //bytes

    //more global variables, constructor

    public void int2byte(int [] src, int off, int len,
        byte [] dst, int off2) {

        if(len < (THRESHOLD / INT_SIZE)) {
            super.int2byte(src, off, len, dst, off2);
            return;
        }

        while(len > 0) {
            int chunkSize = Math.min(BUFFER_SIZE / INT_SIZE, len);

            intBuffer.clear();
            intBuffer.put(src, off, chunkSize);

            byteBuffer.position(0).limit(chunkSize * INT_SIZE);
            byteBuffer.get(dst, off2, chunkSize * INT_SIZE);

            len -= chunkSize;
            off += chunkSize;
            off2 += chunkSize * INT_SIZE;
        }
    }
}

```

Figure 4.4: Hybrid conversion class

very suited for the conversion of small pieces of data. Therefore a final type of conversion called Hybrid was implemented. Hybrid conversion uses simple conversion for very small pieces of data and the NIO/Chunk implementation for large arrays. Figure 4.4 shows the *int2byte* function for it. The constructor for the class is identical to the one used for both NIO implementations.

Chapter 5

NioIbis: a new ibis implementation

As a next step in trying to provide faster, more scalable communication in Ibis we have created a new Ibis implementation, called *NioIbis*. Designed from the ground up with NIO in mind it is possible to use some of the more advanced features of NIO such as the select mechanism. This should make it more scalable than other Ibis implementations. NioIbis uses NIO channels as a transport medium. Although usable for any NIO implementation available, it currently only has support for the TCP implementation. This makes it directly comparable with TcpIbis.

Like every Ibis implementation, the goal of NioIbis is to get messages from a send port to a receive port. Data put into a *NioWriteMessage* is serialized and put onto the network, de-serialized by the receiver, and handed to the user as a *NioReadMessage*. Because there is more than one method of sending and receiving data available in NIO, NioIbis has three different implementations for both send and receive ports. All implementations use the same protocol when sending out and receiving data, so every send port implementation can be combined with every receive port implementation.

Support for the *boolean* primitive type of the Java language is not included in NIO. In NioIbis this problem is solved by always converting a boolean to a single byte (0 or 1), and treating it as any other *byte*. In the rest of this chapter we will use the term "primitive types" to denote all the primitive types *except* the boolean type unless explicitly stated otherwise.

5.1 Serialization

To make NioIbis as efficient as possible, it uses a modified version of the Ibis serialization. Data is written to a *SendBuffer* instead of collecting it into arrays.

When a *SendBuffer* is full, it is given to the send port to be sent out over the network. When a serialization stream is *flushed* the current *SendBuffer* is sent, and the send port is given the command to flush its data as well.

To make the use of these send buffers efficient, a cache is kept with empty buffers. Serialization takes buffers out of the cache when it needs one, and send port implementations return a buffer for recycling when they have completed

sending a `SendBuffer`. It is possible to *duplicate* a send buffer, which will duplicate all data contained in the send buffer. *duplicate* does not actually copy the data, it will only create references to it. When duplicates are returned to the cache, the data will automatically be released once the final copy has been recycled. Duplication of a buffer is useful when doing one-to-many communication.

Internally, the `SendBuffer` object uses an array of NIO `Buffer` objects. The first buffer in the array acts as a header. It holds information about the byte order of the data, the number of elements of each primitive type in the `SendBuffer`, and the length of the footer. Next in the array are `ByteBuffer`s for every primitive type, used to actually store the data. They are sorted by data size of the primitive types, largest first. This order of sending is done to assist receiving, discussed below. Of every byte buffer a view exists in its primitive type used to fill that byte buffer with data. The last buffer in the array is a footer. It is used for padding and is adjusted in size so that the total amount of data in the entire array is a multiple of 8 bytes. Figure 5.1 shows an example of a `SendBuffer` which is ready to be sent. Data is always sent in the native byte order of the sender. If necessary, the receiver performs byte swapping. This way, no unnecessary work is done if their byte order is the same.

At the receiving end, data is received by a receive port as a stream of bytes and is put into a buffer. De-serialization starts by looking at the header. The first byte of the header contains the byte order of the sender, and the buffers are reinitialized if necessary to match it. The byte order of data received will now be automatically adjusted to that of the receiver.

Views of the buffer used for receiving data exist for every primitive type. Figure 5.2 shows the receive buffer and the views. Views span the entire receive buffer, so data of every type can be extracted from anywhere in the buffer. Since the data is sent in descending order of data type size, data is aligned in every view. Moreover, the length of an entire packet is always a multiple of eight, so the next packet which is received is also aligned, and so on. After setting the positions and limits of all views to point to the right pieces of data, the buffer is drained by the serialization mechanism and the data is made available through the `ReadMessage` interface.

Since the incoming data is a stream, and the buffer has a finite size, eventually the last byte of the buffer will be filled. Receive ports automatically start receiving at the beginning of the buffer again when they reach the end. Since the buffer size is a multiple of eight, this will still keep all data aligned. It is possible, though, that data will become *wrapped* around the end of the buffer, making it difficult to efficiently get the data from the buffer. To solve this problem, receive ports will not fill a buffer to its total capacity but will leave some empty space at the end of the receive buffer. When a view of a certain primitive type of which data is wrapped is set up, the wrapped part of the data at the beginning of the buffer is copied into the empty space at the end of the buffer. Now the data is one continuous piece of buffer, so the view can be set up.

5.2 The Send/Receive Thread

A big factor in the performance of any Ibis is the number of threads it needs to handle communication. This is especially true for `TcpIbis`, where every in-

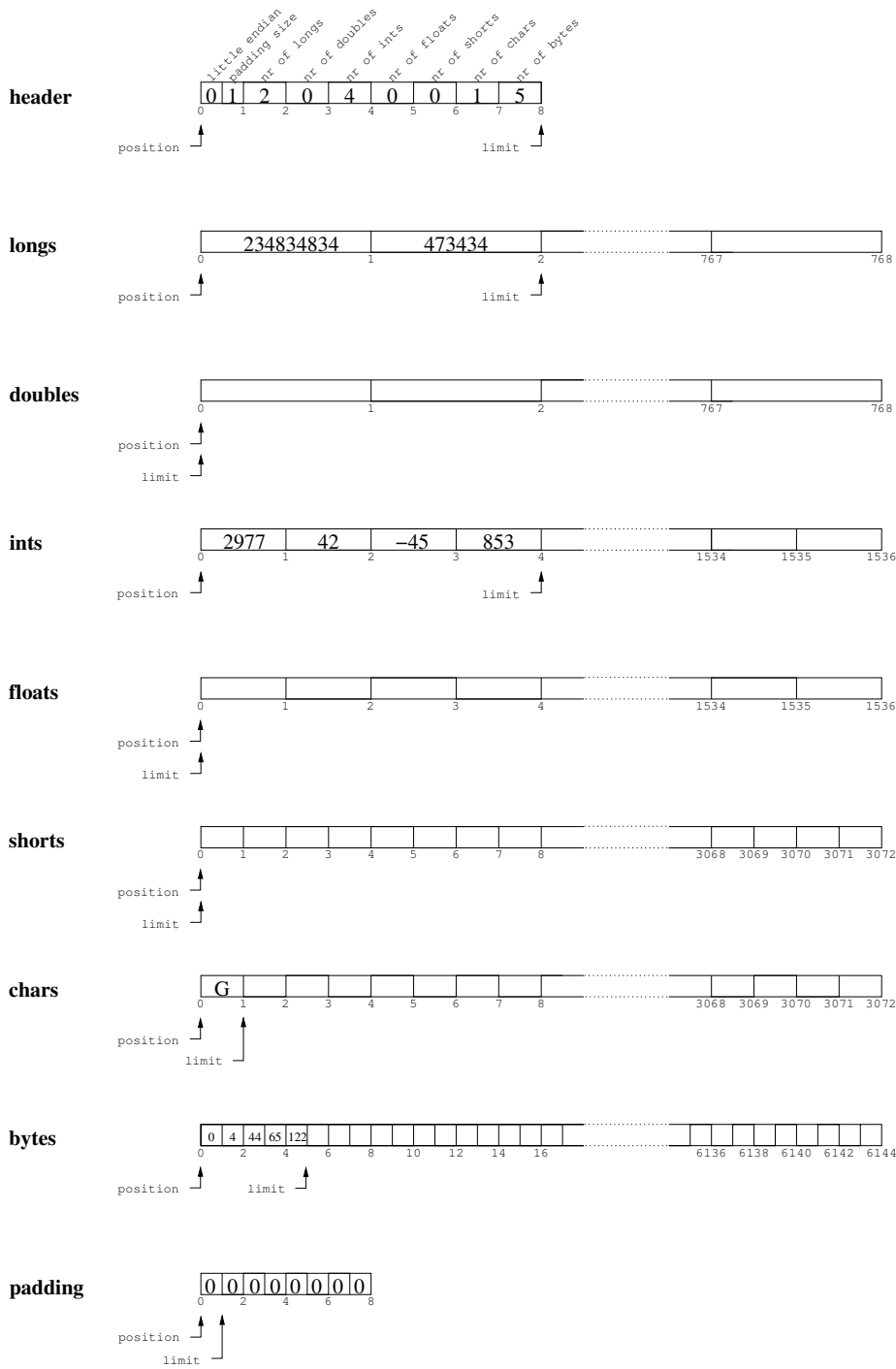


Figure 5.1: SendBuffer ready to be sent

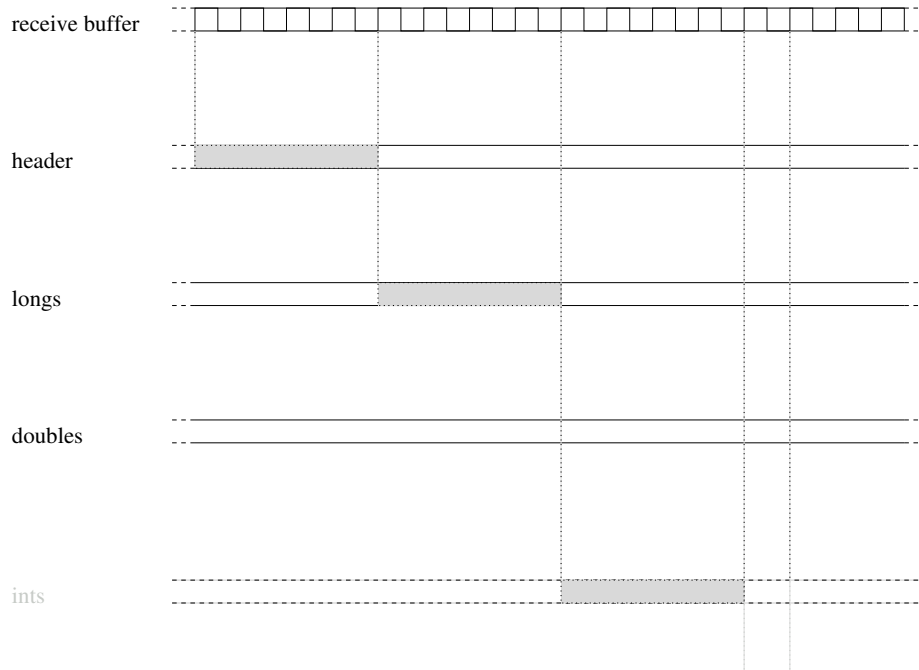


Figure 5.2: Receive buffer ready to be drained

coming connection requires a separate thread to handle communication for that connection. In NioIbis it is possible to handle sending and receiving of *all* data for *every* send and receive port in the entire Ibis instance in one single thread. The usage of a single *send/receive thread* improves the scalability of NioIbis considerably.

The only purpose of the send/receive thread is to continuously look for channels which are ready to transfer data. Channels which need to be monitored by the thread must be registered. After registering, the thread must be signaled when the send or receive port wants to transfer data to or from the channel. The thread will do an upcall to the port which registered the channel when the channel is ready to actually transfer data.

5.3 Thread Overview

Since it is possible to do selection of channels with NIO, the usage of threads in NioIbis differs significantly from TcpIbis. Because TcpIbis only has blocking communication available, every receive port needs to have a separate thread per connection from a send port to wait for incoming messages. In NioIbis, the select mechanism can be used to reduce the number of threads needed to at most one per receive port. An instantiation of NioIbis uses the following threads:

- A single thread for handling new incoming connections from remote send ports to local receive ports. This thread listens to a TCP port for new connections from peer NioIbis implementations, handles the connection set

up, notifies the user of the new connections, and gives new connections to the appropriate receive port.

- A single send/receive thread.
- One or more threads per receive port if upcalls are enabled. This thread continuously calls the *receive* function of a receive port, and generates an upcall for every message it receives.

5.4 NioIbis Send Ports

This section describes the different types of send ports available in NioIbis. Each send port uses a different type of communication. Send ports are given data by the serialization mechanism in the form of a *send buffer*.

The abstract *NioSendPort* class implements the common parts of all send port implementations. It handles new connections, keeps track of all new and lost connections for the user, does serialization, creates new messages, etc.

All implementations of the *NioSendPort* class only implement two functions. One, *doSend*, sends out the given send buffer. It is not mandatory to send out all data immediately. The second function, *flush*, tells the implementation to make sure all the data given to the port so far has actually been sent.

The Blocking Send Port

The *Blocking* send port implementation is the simplest of all three send port implementations. It uses blocking communication to send out data. This behavior is not unlike the *java.io* package, and therefore this send port implementation works much the same way as the *TcpIbis* send port implementation. After getting the buffers that need to be sent from the serialization code these buffers are sent out immediately to the receiver. If there is more than one receiver the buffers are sent to each receiver separately, one after the other. If a receiver is not able to receive data for any reason, sending will stall until the receiver is able to continue. The *flush* function of this implementation does nothing as data is always sent immediately.

The Non Blocking Send Port

The second send port implementation in NioIbis is the *Non Blocking* implementation. It uses non blocking communication to send out data. With this type of communication an attempt to send out data may result in no data being sent at all. The non blocking send port implementation is able to postpone sending data to take advantage of this behavior. It will only block until sending has finished completely when a *flush* is done.

When data is given to this send port the non blocking send port implementation first adds the *SendBuffer* to a list of *pending* buffers. Each outgoing connection has its own pending list. A multicast consists of duplicating the send buffer and adding one copy of it to every list. After updating the pending buffer lists the send port sends out data from the pending buffers list of each connection. The send function will keep sending out data of a connection until it fails to send out a send buffer, or the pending list for that connection is

completely drained. Every send buffer which is send successfully in this manner is returned to the buffer cache for recycling. After trying to send out data to each connection the send function returns, with data possibly remaining in the pending lists.

The *flush* function *does* make sure all data is sent out before returning. It first tries to send out data to each connection with data pending using the same method as described above. It makes a note of every connection which, after this extra sending attempt, still has data pending. Using the select mechanism available in NIO it then waits for one of the noted connections to signal it is ready for sending out more data, and tries to send out its data once more. Once there are no more connections left to wait for all data is sent, and the flush is completed.

The Thread Send Port

The last implementation of the *SendPort* class is the *Thread* send port. It uses the separate send/receive thread to send data. This implementation never sends any data when the *doSend* function is called. Instead, it adds the buffer to the list of pending buffers of one or more connections. Each outgoing connection has its own *Connection* object. A multicast simply adds a duplicate of the send buffer to each of the connections. It then signals the send/receive thread that these connections are now ready to send data. When the send/receive thread detects one of the connections is ready to send out data, it calls a function in the *Connection* object to send the data.

The *flush* function of the thread checks to see if the pending list is empty for each of the outgoing connections. For each connection which has data left to send it does a *wait* until all data has been sent. The send/receive thread wakes the connection when sending is done.

5.5 NioIbis Receive Ports

The NioIbis receive ports are structured in much the same way as the send ports. An abstract *NioReceivePort* class implements all the common functionality, such as connection and message management, and three different implementations take care of the actual communication.

NioIbis supports many-to-one communication. It therefore is possible a receive port is connected to more than one send port. Since only one message is alive at one time in Ibis, a receive port must select an incoming connection to receive a message from every time a new message is needed. The abstract *NioReceivePort* class implements a mechanism to make sure only one message is alive at any time using synchronized methods and delegates the connection selection to the implementations.

The functionality all receive port implementations need to implement consists of two parts. Firstly, they need to implement a *connection selection* mechanism, so the receive port is able to find a connection with a message waiting. Secondly, it should be able to fill a buffer with received data, possibly specifying a minimum amount of data that needs to be received.

The Blocking Receive Port

The first implementation of the *ReceivePort* class is the *Blocking* receive port. It uses blocking communication. This implementation, like the blocking send port, behaves in much the same way as the *TcpIbis* implementation.

With blocking communication, there is no way of selecting from multiple connections simultaneously with only one thread. If data needs to be received from more than one connection, a separate thread per connection must be used to wait for data. *TcpIbis* uses this approach. Since in *NioIbis* the other receive port implementations are able to receive data from multiple connections with only one thread the blocking receive port implementation is not used when multiple incoming connections are needed to a single receive port.

The connection selection mechanism of the blocking receive port implementation simply returns the single connection to the port. When the blocking receive port implementation is asked to receive data it repeatedly does a blocking receive on the channel associated with the port until the required amount of data has been received.

The Non Blocking Receive Port

The non blocking send port implementation in *NioIbis* always has all channels in non blocking mode. Every channel is registered with a single *Selector* object. Selection of these channels therefore can be done with minimal overhead.

The connection selection mechanism of the non blocking receive port works as follows. Firstly, all connections are checked for any waiting messages. If none are found a *select* is done on all the connections to the receive port. For each connection that has been selected, a data receive is initiated. After this, the connections are again checked to see if a message is waiting. This loop continues until either a message is found, or the timeout given by the user expires.

The non blocking receive port has an optimization when selecting a connection. Since all communication is non-blocking, it is possible to optimistically do a receive without the possibility that the call blocks. When a new message is needed, and there is only one connection, a single receive is done on that connection. If successful, no *select* is required.

When data is requested from a specific connection, it is not possible to receive data in a loop until we are done receiving, like the blocking receive port implementation does. Since we have non blocking communication, there is no guarantee any data will be received, and such a loop might increase the load on the system CPU dramatically. So, whenever data is needed, a *select* is done on the specified connection, to make sure data is available. Next, data is received. The process is repeated if not enough data is available yet. As an optimization to this, a single receive is done at the very start of this function, without checking if any data is available. If data was waiting already, this will mean we won't have to do a *select*.

The Thread Receive Port

The last receive port implementation is the *Thread* receive port. It uses the send/receive thread to receive data. When a new connection is added to this receive port, it is registered with the send/receive thread and receiving of data

is enabled. This will make the send/receive thread start checking for incoming data on this connection. When the send/receive port detects data is ready, it signals the receive port data must be received. The receive port then receives data in a buffer for that connection.

The connection selection mechanism of the thread receive port is not actually able to receive any data. One way of handling the selection mechanism would be to repeatedly check each connection if it had any message waiting. This would, however, take a long time when a lot of connections are present, and only a few actually have data ready. Since the main purpose of this implementation is to try to scale better, this is not an option. So, instead, whenever data is received into a buffer which was previously empty, that connection is added to a list. The connection selection mechanism then checks the list of connections and looks if they have a message ready. If no connection has a message ready the function will do a *wait*. If a receive adds a connection to the list, the receive port will be woken up.

If a connection of a thread receive port is asked for data it first checks if any data is available in its buffer. If not, it issues a *wait* until data does become available. It will be woken up by the send/receive thread when data is received.

Chapter 6

Experimental Results

This chapter describes the performance of the Ibis system resulting from the modifications described in the previous two chapters. Both low and high level benchmark programs are used. All the benchmarks are run using the IBM 1.4.1 JVM. When speedups are calculated they are always based on the run time of the benchmark on 1 dual processor node using TcpIbis with simple conversion. We use TcpIbis using simple conversion as a basis when calculating since we used this version as a basis to develop the new conversion types and NioIbis.

The testbed we used for the performance evaluation of these programs is the DAS-2¹ system. DAS-2 is a wide-area distributed computer of 200 Dual Pentium-III nodes. The machine is built out of five clusters of workstations, which are interconnected by SurfNet, the Dutch academic Internet backbone for wide-area communication, whereas Myrinet is used for local communication. For this thesis we only used the cluster at the Vrije Universiteit(VU).

The cluster at the Vrije Universiteit contains 72 nodes. The operating system we used is RedHat Linux 7.2 (kernel version 2.4.18). Each node contains two 1 GHz Pentium-IIIs with 256 Kbyte level II cache and at least 1 GByte RAM, a local IDE disk of least 20 GByte and a Myrinet interface card as well as a Fast Ethernet interface. Since there is no implementation of NIO for the Myrinet network, we will only be using the Fast Ethernet.

6.1 Conversion

Table 6.1 shows the result of a conversion benchmark. This benchmark measures the throughput of the various *Conversion* classes available in Ibis. The table contains results for conversion to and from little endian byte order, the native order of our Pentium III platform. The benchmark was run with arrays of integers, longs and doubles of three different sizes.

Simple Conversion, the original conversion implementation of Ibis, is the first conversion type tested. It performs roughly the same for each of the three array sizes. This is not surprising as it converts elements of an array one by one and is therefore more or less independent of the array size.

The two *NIO Conversion* versions have similar performance for both small and intermediate size arrays. The overhead of using the buffer interface hinders

¹<http://www.cs.vu.nl/das2>

Conversion Type	Simple		NIO				Hybrid	
	write	read	Wrap		Chunk		write	read
			write	read	write	read		
64 byte int[]	167.4	223.0	57.9	57.6	57.0	57.1	143.7	221.5
64 byte long[]	150.4	209.5	57.6	57.6	57.1	56.9	149.0	197.5
64 byte double[]	43.2	82.7	57.6	58.0	57.1	57.1	57.0	57.0
8KB int[]	189.1	265.3	886.6	835.3	884.9	836.6	882.5	841.4
8KB long[]	176.2	250.7	837.9	889.9	838.1	891.4	838.1	881.1
8KB double[]	44.5	88.2	857.9	853.4	860.1	859.1	858.9	857.1
100K int[]	171.9	210.7	57.2	54.1	462.8	471.6	429.8	412.1
100K long[]	164.9	208.2	48.6	56.0	445.8	422.8	447.8	435.0
100K double[]	43.8	81.2	38.2	42.2	372.6	350.8	591.4	547.8

Table 6.1: Conversion Throughput (Mbytes/s) for three different Conversion types

Conversion Type	Simple		NIO				Hybrid	
	write	read	Wrap		Chunk		write	read
			write	read	write	read		
100KB byte[]	∞	117.4	∞	115.9	∞	113.5	∞	118.1
100KB int[]	184.6	90.5	648.2	105.4	643.5	104.8	641.9	104.3
100KB long[]	165.2	78.7	636.4	96.1	620.3	94.8	623.8	96.5
100KB double[]	42.5	51.7	638.1	97.1	627.0	95.5	649.5	96.7
1023 node tree user	37.9	30.2	47.0	31.9	46.0	32.0	48.0	32.6
1023 node tree total	54.7	45.6	71.1	48.2	69.5	48.3	72.6	49.3

Table 6.2: Ibis Serialization Throughput (Mbytes/s) for three different Conversion types

the NIO performance for very small arrays. As a result, Simple Conversion is much faster when used for small arrays. With the somewhat larger 8 KByte arrays, NIO Conversion really shows its strength. With conversion speeds exceeding 800 Mbytes/s conversion is as fast as copying memory. Finally, large arrays performance drops a bit again. This is probably due to caching problems.

There is a big difference between the *Wrap* and the *Chunk* implementations for large arrays. Apparently, wrapping an array and consequently not being able to use a *Direct* buffer is not a very efficient way of converting data. The *Chunk* method performs an extra copy compared to the *Wrap* method, but it still is much faster.

The *Hybrid* conversion implementation shows the expected behavior. For small arrays it gets the same speeds as the Simple conversion type. For medium size and large arrays speeds comparable to the NIO/Chunk implementation are reached. This best of both worlds implementation successfully combines the two conversion techniques and is the best choice when doing conversion of data.

6.2 Serialization

The next benchmark we use to test our modifications to the Ibis system is a *Serialization* benchmark. It tests the throughput for the serialization and deserialization of a number of different object types to and from memory. This includes conversion from primitive types to and from bytes.

Table 6.2 lists the results for the benchmark. We ran the benchmark for a number of different arrays as well as a binary tree of 1023 objects. Each node in the tree has 4 integer values as user data. The table lists two values for

Network Type	100Mbit Ethernet				Loopback			
	Simple	NIO		Hybrid	Simple	NIO		Hybrid
Conversion Type		Wrap	Chunk			Wrap	Chunk	
100K byte[]	10.45	10.39	10.39	10.47	77.34	73.43	76.71	77.38
100K int[]	10.29	10.39	10.43	10.43	41.96	48.86	50.80	50.94
100K long[]	10.41	10.40	10.44	10.48	40.36	49.15	50.83	50.88
100K double[]	10.07	10.41	10.44	10.46	24.84	49.10	50.83	50.88
1023 node tree user	5.60	5.44	5.38	5.24	17.01	16.97	17.76	18.23
1023 node tree total	8.47	8.24	8.14	7.93	25.77	25.67	26.87	27.58
latency	147.6	162.7	164.0	148.4	52.2	52.2	52.6	52.5

Table 6.3: Throughput(Mbytes/s) and Latency(ms) of TcpIbis on 100Mbit Ethernet and the loopback device

Network type	100Mbit Ethernet			Loopback		
	Blocking	Non Blocking	Thread	Blocking	Non Blocking	Thread
100K byte[]	10.38	10.03	8.60	52.08	41.80	59.98
100K int[]	10.46	10.24	8.96	48.69	42.05	25.88
100K long[]	10.44	10.15	8.76	48.77	42.52	25.68
100K double[]	10.44	10.24	9.26	48.46	41.83	25.69
1023 node tree user	5.11	1.17	1.23	13.37	11.38	6.85
1023 node tree total	7.73	1.77	1.86	20.23	17.22	10.36
latency	182.7	203.3	316.3	102.5	129.2	271.8

Table 6.4: Throughput(Mbytes/s) and Latency(ms) of NioIbis on 100Mbit Ethernet and the loopback device

the tree benchmark. One lists the throughput of the user data and one lists the throughput including the overhead needed for serialization. This overhead consists of two pointers per node, one for each child node, encoded by the serialization mechanism as a single integer per pointer. The lower throughput numbers for reading objects as opposed to writing objects is due to the overhead incurred from having to allocate new object when reading.

The benchmark result shows the NIO/Chunk implementation being just as fast as the NIO/Wrap implementation. This is not really surprising since this benchmark never converts arrays larger than 2 kilobytes, and the Chunk and Wrap implementations only differ when used for arrays larger than 8 kilobytes. Consistent with the results of the Conversion benchmark, Hybrid conversion is the best choice. It is almost as fast as the two NIO conversion when used for arrays of data, and the fastest conversion type when used to convert objects.

6.3 Throughput and Latency

The throughput benchmark measures the throughput of an Ibis implementation by sending data from one Ibis instantiation to another, and an acknowledgment back to the sender. Table 6.3 shows the results of this benchmark when run using TcpIbis. The data that is sent consists mostly of arrays of primitive data types. We also tested sending a tree of 1023 objects containing four integers.

On the 100Mbit Fast Ethernet network TcpIbis is able to utilize the network bandwidth fully for every data type. To test TcpIbis to its limits we also ran the benchmark by starting both Ibis instantiations on a single DAS-2 node, using the loopback device for communication. The results (also in Table 6.3) indicate the new conversion classes added to TcpIbis outperform the original conversion

class, listed in this table as Simple conversion.

To compare TcpIbis to NioIbis, we also ran this set of benchmarks on NioIbis. Instead of varying the conversion used, which is done implicitly in NioIbis, we ran the test for different implementations of the send and receive ports. Table 6.4 shows the results. On 100Mbit, the blocking send and receive port implementations performance is on a par with TcpIbis. The Non-Blocking and Thread implementations have a somewhat lower performance. This is mostly due to the fact that the Non-Blocking and Thread implementations are not optimized for throughput, but rather for scalability. With only a single connection, as is the case with this benchmark, using the select mechanism is unnecessary and hinders throughput. The non blocking and thread implementations are therefore only used when one-to-many or many-to-one communication is needed.

On the loopback device, NioIbis shows the same behavior as it does on the 100Mbit network. The blocking implementations performance is almost equal to TcpIbis, while the Non-Blocking and Thread implementations performance is lower due to the usage of the selection mechanism and non blocking communication. The throughput for byte arrays is not as good as TcpIbis in any implementation of NioIbis. This degradation in speed comes from the fact that NioIbis has to copy the data once into a *Buffer* while TcpIbis can simply send the original array. The throughput for object trees is also lower with NioIbis than with TcpIbis. This slowdown is due to the fact that the NIO interface is optimized for large amounts of data. The tree objects are serialized into many separate integers, so multiple calls with a single integer are done to put this data into a *Buffer*. The *Buffer* interface makes writing small objects, or objects consisting of many small elements, quite expensive in NioIbis. If the original Ibis serialization is used with NioIbis, tree throughput is as fast as TcpIbis. The drawback of using the original serialization is an extra copy however, and when used in something other than micro benchmarks this hinders performance considerably.

Also shown in Table 6.3 and Table 6.4 is the latency of TcpIbis and NioIbis. It is clear that the latency of NioIbis is worse than that of TcpIbis. This difference is again due to the fact that the NIO interface is optimized for handling large amounts of data. The effort needed for sending data using NIO is much greater than needed to send out data using the *java.io* interface. Also, the NIO interface is designed with multi threading in mind. Therefore synchronization of methods and data is used throughout NIO. This synchronization also increases latency by a considerable amount. Both the Non-Blocking and the Thread implementation make frequent use of the select mechanism in NIO, and therefore have an increased latency. The Thread implementation is not optimized for low latency at all, and multiple thread switches may be needed to deliver a single message.

Overall the throughput and latency tests show some mixed results. Usage of NIO for conversion purposes increases throughput. Using NIO for the actual sending and receiving of data however, is not always the best choice. When only a single connection is used per thread, the heavyweight NIO interface and mechanisms do not improve, and may even hinder, performance.

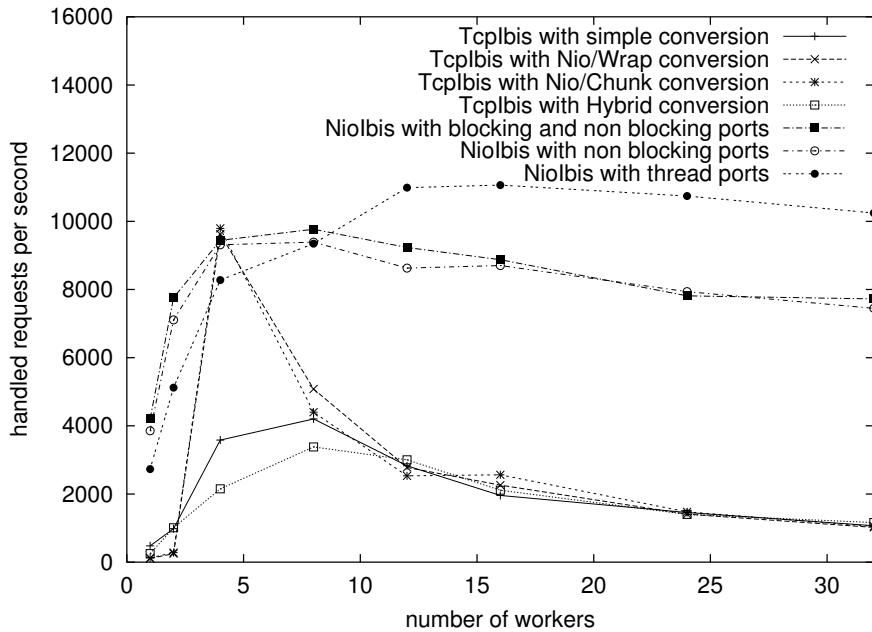


Figure 6.1: Master/Worker benchmark

6.4 A Master/Worker benchmark

The next benchmark we use to evaluate the performance of our modifications is a simulation of a master/worker type program. The workers continuously send requests (in this case every worker simply sends an object identifying itself) to the master. The master responds to each request by sending a data object. The data object contains a single element of each primitive type. Each time the master has handled 100.000 requests the master computes the number of requests handled per second.

In TcpIbis, this benchmark requires the creation of one thread per worker, to wait for new messages to arrive on. This thread is needed because the TcpIbis receive port uses blocking communication, and is thus not able to select from multiple incoming connections with one thread. With NioIbis this is not needed. A single *select* can be done to request the next connection which has a message available.

Figure 6.1 shows the number of requests a master is able to handle per second. For TcpIbis every conversion implementation is tested. For NioIbis, different implementations of send and receive ports are used for each run. The first combination of send and receive port implementations uses as much blocking ports as possible. This means using blocking send ports for all connections. It uses a non blocking receive port at the master for receiving from all the workers and uses blocking receive ports at the workers for receiving the reply from the master. Using a blocking receive port at the master would mean having to handle multiple incoming connections in a single thread, something not possible with blocking communication. The two other runs use only non blocking or

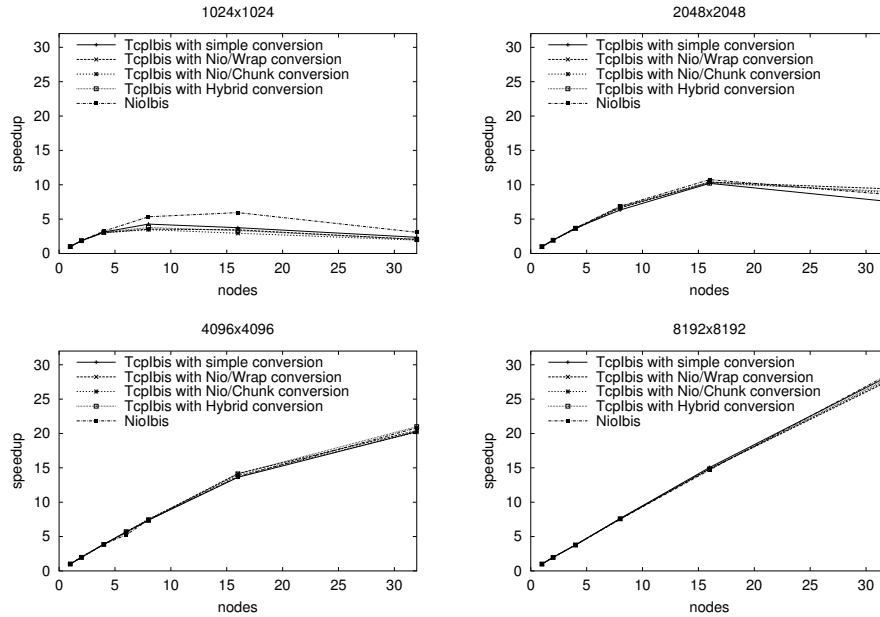


Figure 6.2: SOR with different matrix sizes

thread port implementations for communication.

The results indicate that both implementations become saturated with requests when eight worker nodes are used. NioIbis implementations manage to handle two and a half times as much request as TcpIbis. When even more workers are used, the network traffic starts to overwhelm the master node, and fewer requests are handled per second. TcpIbis has so many problems with handling all the threads and data, that increasing the workers to sixteen halves the number of requests actually handled. NioIbis scales significantly better. Although taking a while to reach full speed the Thread implementation eventually reaches a performance of over 10.000 requests per second.

6.5 Successive Overrelaxation

SOR Red/Black Successive Overrelaxation (SOR) is an iterative method for solving discretized Laplace equations on a matrix. The program distributes the matrix row-wise among the machines. Each machine exchanges one row of the matrix with its neighbors at the beginning of each iteration. We used matrices from 1024x1024 to 8192x8192 as input.

This implementation of SOR terminates once the difference between two successive matrices comes below a certain threshold. To calculate this threshold every node sends the maximum difference between two iterations as a single double value to the master node. The master node calculates the maximum of all received values and broadcasts this value to all nodes.

On TcpIbis, all three communication types, neighbor communication, reduce and broadcast, are done using the same send and receive port implementation. However, on NioIbis different send and receive port implementations are auto-

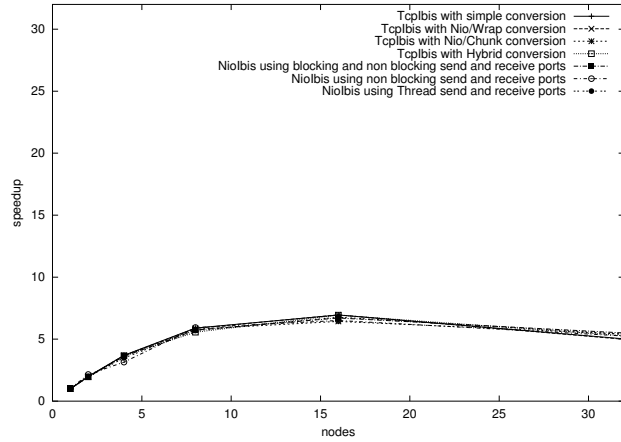


Figure 6.3: Barnes-Hut using the Satin system on Fast Ethernet

matically selected for different communication types. The neighbor one-to-one type communication uses both the blocking send and blocking receive port. The many-to-one reduce operation uses the blocking send and the non-blocking receive port. The non-blocking receive port is used because this implementation is more efficient when used with multiple incoming connections. The one-to-many broadcast communication uses the non-blocking send port and the blocking receive port implementations. The non-blocking send port is used because of its ability to handle multicast better than the blocking version can.

Figure 6.2 shows the speedup of the SOR benchmark with four different matrix sizes. With a 1024x1024 matrix, it clearly shows the benchmark benefits from the optimized versions of the NioIbis send and receive ports. Analysis of the results show the time spent on the reduce step of the algorithm is up to three times faster with NioIbis. With the larger matrices the overhead of communication decreases, so the difference between the different Ibis versions diminishes as well.

6.6 Barnes-Hut

Barnes-Hut is an $O(N \log N)$ N-body simulation. The version [7] used for testing is implemented with the Satin [10] divide-and-conqueror system. At the start of each iteration a tree of data is broadcast from the master node.

Figure 6.3 shows the speedup curve of Barnes-Hut run on both TcpIbis and NioIbis. The three NioIbis lines represent runs with different send and receive port implementations used. The default selection mechanism selects non blocking implementation when many-to-one or one-to-many communication is needed, and blocking implementations otherwise. Two alternative selections were done. One uses only Non-Blocking implementations, one only Thread implementations.

The speedup curve shows almost no performance difference between the different Ibis implementations. Since Barnes-Hut requires large amounts of com-

munication, its performance is bound by the throughput of the Fast Ethernet network on the DAS-2. In general, NioIbis performance is on a par with TcpIbis when used in situations optimal for TcpIbis.

Chapter 7

Conclusions and Future Work

Conclusions

In this thesis we have tried to use the New Java I/O Interface in the Ibis environment to achieve faster, more scalable communication in parallel computing. We have shown it is indeed possible to increase communication speed with NIO. A simple and unobtrusive way of increasing the throughput of communication is possible by using the conversion functionality built into NIO. With more work, it is also possible to use a selection mechanism. This increases the scalability of group communication considerably.

Using NIO also has some disadvantages. For one, NIO has a complicated interface. This is a fundamental problem rising from the fact that NIO tries to expose the underlying communication layer as much as possible and flexibility is usually chosen over simplicity. Because of this complexity, NIO is not really suited for handling small amounts of data. Much bookkeeping is required for sending data independent of the size of the data and all operations on channels are thread safe. This leads to inefficiencies when used for latency sensitive programs. Also, the selection mechanism as found in NIO is not very easy to use. Especially when using selection in a multi-threaded environment, where the frequent use of synchronization mechanisms can lead to unexpected deadlocks.

A more serious problem with NIO is the fact that the type of interface NIO uses requires users of the interface to make choices which may impact performance in ways not deductible by looking at the interface alone. For example, there are two possible implementations of a conversion routine with NIO. The difference in performance between these two implementations, described in the thesis as NIO/Wrap and NIO/Chunk conversion, can be as much as a factor of 10.

Another unknown factor when using NIO is the select mechanism. For small numbers of connections using a separate thread per connection might be faster than using the somewhat cumbersome select present in NIO. The break-even point depends entirely on the implementation of the select mechanism by the virtual machine and underlying operating system.

There is no real solution to this system design choice problem. One way

of handling it is to implement different possible methods of doing the same thing, and then choosing the optimal one at run time or by hand. However, this involves much extra code, if possible at all. Another option is to make a best effort to select an appropriate solution, and only implement that single solution. However, if chosen incorrectly this might place a hefty penalty on performance.

As a side effect of our research we created a new Ibis implementation, NioIbis, which scales considerably better than its counterpart TcpIbis. Also, in circumstances less favorable to NioIbis it does not incur a performance penalty compared to TcpIbis.

Future Work

This section describes some extensions of the research described by this thesis.

First, it might be possible to optimize NioIbis further for low latency. A *light weight* NioIbis implementation which only supports point-to-point blocking communication should be able to reduce the latency of NioIbis considerably, at the cost of losing features. Even within the *monolithic* NioIbis some further optimizations are still possible.

Secondly, while programming NioIbis it became apparent that different buffer sizes influence performance greatly. More research should be done to look into the possibility of auto detecting the optimal size of the various buffers used throughout Ibis. Other options, such as the receive port implementation to use, might be detectable somehow as well.

Lastly, the usage of NIO as described in this thesis is limited to the TCP implementation of NIO which ships with Java™1.4. Since the NIO interface is a general interface, it is also possible to create a NIO implementation using some other communication mechanism, such as a Myrinet network. It would then be possible to use NioIbis on this NIO implementation without changing much code in NioIbis itself.

Chapter 8

Acknowledgments

Without the help of a lot of friendly people this thesis would never have existed. I would especially like to thank Henri Bal, Jason Maassen, Rob van Nieuwpoort, Cerial Jacobs, Rutger Hofman, Fabrice Huet, Maik Nijhuis, Olivier Aumage, Kees Verstoep, Kees van Reeuwijk and Marjolein van Gendt for all their guidance, technical help, and moral support.

Bibliography

- [1] H. Bal, R. Bhoedjang, R. Hofman, C. Jacobs, K. Langendoen, T. Ruhl, M.F., and Kaashoek. Performance evaluation of the Orca shared object system. *ACM Transactions on Computer Systems*, 16(1):1–40, Feb. 1998.
- [2] N. Boden, D. Cohen, R. Felderman, A. Kulawik, C. Seitz, J. Seizovic, and W. Su. Myrinet: A gigabit-per-second local area network. *IEEE Micro*, 15(1):29–36, Jan. 1995.
- [3] R. Hitchens. *Java NIO*. O’Reilly, Aug. 2002.
- [4] J. Maassen, T. Kielmann, and H. Bal. GMI: Flexible and Efficient Group Method Invocation for Parallel Programming. In *In proceedings of LCR-02: Sixth Workshop on Languages, Compilers, and Run-time Systems for Scalable Computers*, pages 1–6, Washington DC, March 2002.
- [5] J. Maassen, R. van Nieuwpoort, R. Veldema, H. Bal, T. Kielmann, C. Jacobs, and R. Hofman. Efficient Java RMI for Parallel Programming. *ACM Trans. on Programming Languages and Systems (TOPLAS)*, 23(6):747–775, November 2001.
- [6] Message Passing Interface Forum. MPI: A message-passing interface standard. Technical Report UT-CS-94-230, Message Passing Interface Forum, 1994.
- [7] M. Nijhuis. Divide-and-conquer Barnes-hut implementations. Master’s thesis, Faculty of Sciences, Division of Computer Science, Vrije Universiteit, Amsterdam, The Netherlands, May 2004.
- [8] Sun Microsystems. Guide to NIO in java 1.4. <http://java.sun.com/j2se/1.4/docs/guide/nio/index.html>, 2002.
- [9] R. V. van Nieuwpoort, J. Maassen, R. Hofman, T. Kielmann, and H. E. Bal. Ibis: an efficient java-based grid programming environment. In *Joint ACM Java Grande - ISCOPE 2002 Conference*, pages 18–27, Nov. 2002.
- [10] R. V. van Nieuwpoort, J. Maassen, G. Wrzesinska, T. Kielmann, and H. E. Bal. Satin: Simple and efficient java-based grid programming. *Journal of Parallel and Distributed Computing Practices*, 0(0):0, 2004.