latteMPI: Constructing a Message Passing Interface Implementation in Java

Author:
David Alexander BECKINGSALE

University Number:
0814366

Supervisor:
Professor Stephen JARVIS

May 2011
ABSTRACT

The latteMPI project is a pure Java implementation of a subset of the Message Passing Interface (MPI) Standard. The MPI Standard defines a number of methods that facilitate writing programs using the message-passing model of computing, where processes share data using explicit calls to send and receive functions. Performance of parallel programs running on both latteMPI and a C implementation of the MPI Standard is investigated. The performance of latteMPI is slower than the equivalent C implementation, but some Java-specific advantages are discussed: reduction of setup time and portability of programs.

Keywords High Performance Computing, Java, Message Passing, MPI, Parallel Programming.
ACKNOWLEDGMENTS

A number of people deserve thanks for the considerable support they have offered me throughout the course of my project.

My friends: Robert Bird, for his encouragement and reminding me when to take a break, and Richard Bunt, for the enlightening discussions we’ve had regarding MPI and our respective projects.

I would like to thank my supervisor, Stephen Jarvis for everything he’s done this year, least for all supervising my project.

Simon Hammond, I would like to thank for his expertise and support this year, as well as his remarkable patience answering all my – sometimes silly – questions.

Matthew Leeke, for his continued support and encouragement, not only this year, but for the past 3 years. Without his help proofreading this report, I’m sure it would be far less polished.

Finally, I would like to thank Jordan for all her support this year, I’m sure she’s learnt more about MPI than she ever wanted to.
CONTENTS

Abstract iii
Acknowledgments v
List of Figures xi
List of Tables xiii

1 Introduction 1
   1.1 Message Passing . . . . . . . . . . . . . . . . . . . . . . . . . . . . 2
   1.2 The MPI Standard . . . . . . . . . . . . . . . . . . . . . . . . . . 2
   1.3 Java . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 3
       1.3.1 Java and HPC . . . . . . . . . . . . . . . . . . . . . . . . . 5
   1.4 Motivations . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 6
   1.5 Report Summary . . . . . . . . . . . . . . . . . . . . . . . . . . . 7

2 Related Work 9
   2.1 MPI Implementations . . . . . . . . . . . . . . . . . . . . . . . . . 9
   2.2 Java MPI Implementations . . . . . . . . . . . . . . . . . . . . . . 10
       2.2.1 mpiJava . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 10
       2.2.2 MPJ Express . . . . . . . . . . . . . . . . . . . . . . . . . . . 14
   2.3 Applications of MPI . . . . . . . . . . . . . . . . . . . . . . . . . . 14
       2.3.1 Scientific and Research Based Applications . . . . . . . . . 14
       2.3.2 The Gordon Bell Prize . . . . . . . . . . . . . . . . . . . . . 15
# 11 Conclusion

11.1 Project Context ................................................. 95
11.2 Performance Summary ................................. 95
11.3 Project Experiences .............................. 96
11.4 Closing Remarks ................................... 96

## Bibliography

97

## Appendices

101

### A Specification

A.1 Problem ..................................................... 103
A.2 Objectives .................................................. 103
A.3 Methods ...................................................... 104
  A.3.1 Design .................................................. 104
  A.3.2 Implementation ..................................... 104
  A.3.3 Performance Analysis .............................. 105
  A.3.4 Project Report ...................................... 105
A.4 Timetable .................................................... 106
A.5 Resources .................................................... 106
A.6 Legal, Social, Ethical, and Professional Issues ......... 107
  A.6.1 Social and Professional ......................... 107

### B User Guide

B.1 Setup .......................................................... 109
  B.1.1 Environment ........................................ 109
  B.1.2 Compilation ......................................... 110
B.2 Programming with latteMPI ............................ 110
  B.2.1 Compilation ......................................... 112
  B.2.2 Running programs ................................ 112
B.3 Further Information ..................................... 113

### C Presentation Slides

115
## LIST OF FIGURES

1.1 Overview of the message passing model. ........................... 2  
1.2 “Hello World!” in Java (hw.java) ................................. 4  
1.3 The Java byte-code for hw.java ................................. 4  
1.4 Java compilation and use. ........................................... 5  
2.1 Using JNI to call native code ........................................ 11  
2.2 Java send method - Comm.java .................................. 12  
2.3 Native send method - mpi_Comm.c .......................... 13  
4.1 Abstract diagram of the system. .................................... 26  
4.2 Principal classes of the system. .................................... 32  
5.1 Components during the initialisation of the system. .......... 36  
5.2 System objects on initialisation. ................................. 38  
5.3 NetDevice – low level communication. .......................... 39  
5.4 The Send operation. ................................................. 40  
5.5 Message travelling through the system layers during the Send operation. ................................................. 41  
5.6 The Recv operation. ................................................. 42  
5.7 Message travelling through the system layers during the Recv operation. ................................................. 44  
5.8 A simple broadcast operation. ..................................... 45  
5.9 The Bcast operation. ................................................. 45  
5.10 Tree broadcast for 6 processes, rooted at node 0. .......... 46  
5.11 A simple reduce operation. ...................................... 48  
5.12 The Reduce operation. .......................................... 48
5.13 Tree reduce for 8 processes, rooted at node 0.  
5.14 A simple allreduce operation.  
5.15 The AllReduce operation.  
5.16 The Barrier operation.  
6.1 The testInt method – unit test.  
6.2 Output of running unit tests on the latteMPI code.  
7.1 Graph of results for ping-pong benchmark.  
7.2 Results of the ping-pong benchmarks as two separate graphs.  
7.3 Graph of results for Floyd-Warshall 1000 node graph benchmark.  
7.4 Graph of results for Floyd-Warshall 5000 node graph benchmark.  
7.5 Example input file for SWEEP3D  
7.6 Graph of results for SWEEP3D benchmark using problem size 50^3.  
7.7 Graph of results for SWEEP3D benchmark using problem size 75^3.  
7.8 Graph of results for SWEEP3D benchmark using problem size 100^3.  
7.9 Graph of results for SWEEP3D benchmark using problem size 250^3.  
7.10 Graph of results for SWEEP3D benchmark using problem size 500^3.  
8.1 Term 1 Gantt chart.  
8.2 Gantt chart for the implementation phase.  
8.3 Job submission script used for the SWEEP3D program.  
A.1 Project timeline  
B.1 Simple “Hello World” program using MPI.
### LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1</td>
<td>Minimal subset of MPI operations</td>
<td>28</td>
</tr>
<tr>
<td>4.2</td>
<td>Minimal subset of non-blocking MPI operations</td>
<td>28</td>
</tr>
<tr>
<td>5.1</td>
<td>Concrete MpiOps provided by latteMPI.</td>
<td>49</td>
</tr>
<tr>
<td>6.1</td>
<td>Programs used for system testing.</td>
<td>59</td>
</tr>
<tr>
<td>7.1</td>
<td>Table of results for the ping-pong benchmark. Sizes detailed are $2^n$ increments for brevity.</td>
<td>66</td>
</tr>
<tr>
<td>7.2</td>
<td>Time taken to convert doubles to/from bytes.</td>
<td>67</td>
</tr>
<tr>
<td>7.3</td>
<td>Table of runtimes for the Floyd-Warshall program.</td>
<td>68</td>
</tr>
<tr>
<td>7.4</td>
<td>Table of runtimes for SWEEP3D on input size $100^3$.</td>
<td>72</td>
</tr>
<tr>
<td>8.1</td>
<td>Project Timetable: Term 1.</td>
<td>78</td>
</tr>
<tr>
<td>8.2</td>
<td>Project Timetable: Implementation Phase</td>
<td>86</td>
</tr>
<tr>
<td>A.1</td>
<td>Project Timetable</td>
<td>108</td>
</tr>
</tbody>
</table>
CHAPTER 1

INTRODUCTION

High-performance computers have become a permanent fixture in the computing landscape, and as their popularity has grown, so has the demand for computational power. Shared memory computers that were popular in the past have been replaced by new distributed memory machines – many connected processors with their own memory – in order to maximise performance. Shared memory machines with comparable performance are prohibitively expensive, so distributed memory machines occupy a price/performance sweet spot.

One of the most popular forms of distributed memory machine is the cluster. A typical cluster consists of a number of commodity computers connected via a high performance network. The cluster is a popular choice in the high-performance computing domain because of the low price and high availability of parts.

Distributed memory bought a new problem to high-performance computing; problems needed to be decomposed in order to run on each of the separate processors. When problems are broken down over multiple processors, there comes the need for these processors to share data. Since the processors in distributed memory machines do not share memory, sending messages becomes the ubiquitous way to share data.

Message passing is a vital part of programming high-performance computers. In the very early days of distributed computing the Parallel Virtual Machine (PVM) provided a way to run programs on large numbers of distributed processors, utilising message passing for communication. However, a
CHAPTER 1. INTRODUCTION

1.2 The Message Passing Interface Standard

The Message Passing Interface (MPI) Standard is a message-passing library interface specification. The MPI Standard is written by the MPI Forum and
presents a set of functions that aid programming in the message passing style. The functions presented in the MPI Standard make it easy for programmers to use the message-passing model of parallel programming.

The MPI Forum is the group responsible for creating and developing the MPI Standard. The Forum contains members from all parts of the computing community. Academic members are joined by industry representatives and together the members develop and maintain the MPI Standard through frequent meetings and online discussion.

Whilst the MPI Standard is commonly encountered anywhere message-passing is used, it is still only a standard, not a concrete implementation. Various implementations of the MPI Standard exist for C, C++ and Fortran (the languages which currently have bindings defined in the standard). The bindings for C++ have recently been deprecated, but they are still provided by many implementations. Open MPI [15] and MPICH [25] are open-source implementations originating from Indiana University and Argonne National Laboratory respectively. High-performance computing vendors also produce implementations tuned to the hardware that they provide, examples of these include Cray MPI and the Intel MPI Library.

The MPI Standard describes only the calling structures required without adding any constraints on how the messages should be processed. By focusing on this, the semantics of message passing, vendors are permitted to customise the low-level behaviour of MPI to better fit their architectures and platforms.

Rather than a whole new programming language, the MPI Standard presents a set of functions that support the message passing paradigm. These functions are called from the user program and allow communication with the remote processes. More information on programming using MPI is presented in [16] and [31].

1.3 Java

Java is an imperative programming language that has been under active development since 1991. Java is developed by Sun Microsystems (now a subsidiary of Oracle) although there are a number of open source and community influences on its direction. The Java programming language is similar in style and syntax to C++, also employing the object-oriented paradigm. The prominent feature of Java is the fact that the source code is compiled to byte-code, which is then run on the Java Virtual Machine (JVM).
public class hw {
    public static void main (String[] args) {
        System.out.println("Hello World!");
    }
}

Figure 1.2: “Hello World!” in Java (hw.java)

public class hw extends java.lang.Object{
public hw();
    Code:
    0: aload_0
    //Method java/lang/Object."<init>"():V
    1: invokevirtual #1;
    4: return

public static void main(java.lang.String[]);
    Code:
    //Field java/lang/System.out:Ljava/io/PrintStream;
    0: getstatic #2;
    //String Hello World
    3: ldc #3;
    //Method java/io/PrintStream.println:(Ljava/lang/String;)V
    5: invokevirtual #4;
    8: return
}

Figure 1.3: The Java byte-code for hw.java

Figures 1.2 and 1.3 show the transformation from source code to byte-code. The byte-code is somewhat similar to assembly language. After the source code is compiled to byte-code, it can be run on any implementation of the JVM.

It is important to note the distinction between the Java programming language and the Java Virtual Machine. The Java programming language is generally only associated with the JVM, but can be converted to native machine code by compiling it with a different compiler. Conversely, other languages can be compiled to byte-code so that they can run on the JVM. Figure 1.4 shows this visually. Java code is run through the Java compiler to generate the byte-code which can then run on a JVM on any hardware.
It is beyond the scope of this report to explain Java in more technical detail, and from now on a grasp of Java syntax and a basic understanding of the JVM is assumed. For additional information on programming with Java, the reader is referred to [30] and [24].

Java is a popular programming language, and is holding onto its popularity despite its age and the current developments in the Java community. The TIOBE Index [38] records the popularity of programming languages. The rankings are calculated by conducting a search of the form +"<language> programming" on some of the most popular search engines. The number or results is normalised over the top 50 languages and then presented as a table. Java is the top programming language as of January 2011. The TIOBE index provides a fairly accurate index of general programming language popularity. Although the data is based entirely on search engine results, this is apt for programming language popularity because popular programming languages generate vast online communities. The size of these communities will generate an appropriate proportion of search engine ‘hits’, more hits for the more popular languages.

1.3.1 Java and High-Performance Computing

Of particular interest to this project is Java in a high-performance computing environment. Is Java a viable choice for performing intensive calculations where speed and accuracy are of particular importance?

Smith and Bull [36] provide an overview of Java in the context of Scientific Computing, highlighting several advantages of Java, including: portability, network centricity, and programmer availability. Also described is the Java Grande Forum (JGF), an initiative concerned with Java applications that have
large memory or processing power requirements.

The SciMark 2.0 benchmark [34] is a Java benchmark that measures the performance of numerical codes\(^1\) on the JVM. It is also incorporated into the efforts of the JGF. The results of these benchmarks are measured in floating point operations per second (FLOPS), the common standard for measuring performance of computers, particularly for numerical applications.

The benchmarks used by SciMark use a number of common numerical tasks to measure performance, allowing them to provide a real-world measure of JVM performance. The SciMark results are collected on [34], and analysis of these results shows that the performance of Java codes is strongly linked to the JVM implementation they are running on. The performance of the benchmarks increases measurably with each new JVM version [8].

As alluded to in Section 1.3, the real advantage of Java is its JVM architecture. By keeping the code and the execution environment separate the performance of Java codes can improve every time the JVM is upgraded.

The JGF Concurrency and Applications Benchmark group produces benchmarks in order to measure the performance of various Java implementations and their native counterparts. Preliminary tests with these benchmarks help refute the claims that Java is slow, showing minimal runtime performance differences between C and Java [36].

### 1.4 Motivations

High-performance computing is an important research area for Computer Science. Message passing is at the heart of programming for distributed memory computers, and a Java implementation of the Message Passing Interface will provide a number of benefits.

- Java byte-code is portable between systems. A program written in Java can be compiled to byte-code on one machine and then run on any other JVM.

- The performance of the JVM continues to improve with every new version, a Java implementation of MPI will benefit from all these improvements with no extra work.

- There is currently no standard MPI implementation for the Java platform.

\(^1\)In this context codes refers to the application, rather than the source code.
• To allow comparison of C and Java applications in a message-passing context.

1.5 Report Summary

This chapter focuses on introducing the MPI Standard and providing a brief introduction to the context of the problem. Chapter 2 describes the research undertaken throughout the project; first documenting MPI implementations, then looking at applications using MPI. Chapter 3 documents the system specification, in terms of functional and non-functional requirements, as well as discussing the legal, social, ethical, and professional issues that apply to this project. Chapters 4 and 5 present the design and implementation of the project, while Chapter 6 discusses testing the project. Chapter 7 focuses on performance evaluation, containing results of the many performance tests run. Chapter 8 evaluates the project management and provides details of the development process and tools. Chapter 9 is a brief chapter containing a critical evaluation of the project. Chapter 10 contains an evaluation of the project with respect to the functional and non-functional requirements. Finally, Chapter 11 concludes the report.
CHAPTER 2

RELATED WORK

This chapter will highlight the place of latteMPI amongst the previous and current work in this subject area. Focusing on current implementations of the MPI Standard we will look at both Java and C. Also discussed are some applications of MPI: extensions to existing MPI implementations, and scientific programs making use of MPI.

2.1 MPI Implementations

There are many implementations of the MPI Standard for the defined C and Fortran bindings. MPICH [25] was the first of these implementations to be developed. MPICH was implemented in parallel with the development of the MPI Standard, allowing the MPI Forum to see whether the proposed functions were appropriate and feasible. An alternative open-source implementation is Open MPI [15], which focuses on creating a high performance, portable implementation of the Standard.

Most MPI implementations follow a similar design pattern, that of forming the components presented in the Standard into separate modules or objects. MPICH takes particular advantage of this by allowing default modules for a task such as sending data, to be replaced with a hardware or operating system specific version that generally improves performance and reliability. As we will see later, this modular design fits nicely with the object-oriented philosophy of Java, and features heavily in the design of this project.

Given the open source nature of both MPICH and Open MPI, it is natu-
ral that they have formed the basis of my research into MPI implementations. As mentioned previously, vendor specific implementations are provided for a number of major platforms; however, as these are proprietary software they are unsuited to both research and learning. My research has since been directed towards the open source implementations and more abstract material.

The current development version of MPI, MPI 3, is a major revision of the Standard, as suggested by its version number. The current version of the Standard, MPI 2.2 was finalised in 2009, but its parent major revision, MPI 2 was finalised in 1996. The large time span between major releases is attributable to issuing clarifications to the Standard, with these being released more frequently.

2.2 Java and MPI Implementations

Although there are no defined bindings for Java in the MPI Standard, there already exist a number of implementations that provide the required functions using different techniques.

Two implementations of the MPI Standard in Java are mpiJava [10] and MPJ Express [6]. Each implementation takes a different approach to providing the message passing functions. mpiJava uses an existing C implementation, while MPJ Express provides all the functions as pure Java code. These two implementations are closest to the project in terms of their scope as well as their methods. The review of these implementations helped to establish a baseline from which the latteMPI can take some inspiration.

2.2.1 mpiJava

mpiJava takes what may seem like a strange approach to providing the functions defined in the MPI Standard. The functions are provided as a set of JNI wrappers around an existing native MPI implementation.

JNI

The Java Native Interface (JNI) provides a mechanism for Java programs to interact with native code; native code being that which is compiled to the language of the machine, rather than being an interpreted byte-code like Java.

Each JNI function exists in a separate C or C++ file. When this function is called from the Java code the JVM passes it a pointer to a JNIEnv object, as
well as all the function arguments received in Java. The JNI function can use the `env` pointer to communicate with the JVM. Figure 2.1 contains a simple visual representation of this process.

The JNI function can perform anything that could happen in a normal C/C++ function - the main advantage to JNI is that data from the Java program can now be manipulated with hardware or operating system specific features.

From this brief overview of JNI, we can begin to understand how mpiJava uses it to provide a complete and robust implementation of the MPI Standard. Each MPI function is represented as a JNI call, which accesses the underlying MPI implementation. The user program will call the MPI functions as defined by the mpiJava library, mpiJava is then responsible for passing the arguments correctly to the underlying native MPI implementation using JNI. The return values of the native function can then be passed up via JNI to the Java library and then on to the user program.

The listing in Figure 2.2 contains the Java source code for the basic blocking send as provided by mpiJava. If the data is an object, the Send method called by the Java program serializes the data to be sent (the line `if(type.isObject())`), before using the native send method to send first a header and then the data. If the data is primitive (an int value for example) the data is just passed to the native send function. The native keyword instructs Java that this function is provided as a JNI function.

---

1Further explanation of using JNI is beyond the scope of this report, please consult [23] for
public void Send(Object buf, int offset, int count, Datatype type,
  int dest, int tag) throws MPIException {
    if (type.isObject()){
      byte[] byte_buf = Object_Serialize(buf,offset,count,type);
      int[] length_buf = {byte_buf.length, count} ;
      send(length_buf, 0, 2, MPI.INT, dest, tag); // header
      send(byte_buf, 0, byte_buf.length, MPI.BYTE,dest, tag) ;
    } else { send(buf, offset, count, type, dest, tag); }
  }

private native void send(Object buf, int offset, int count, Datatype type,
  int dest, int tag);

Figure 2.2: Java send method - Comm.java

Figure 2.3 presents a listing of the send method that is called from Send. This is the native send method referred to by the Send method in the Java class. Looking at the source code we can see that mpiJava finds the MPI_Comm object (which contains information about the group of communication processes) and the MPI_Datatype of the information to be sent. The #ifdef GC_DOES_PINNING is used to work around a garbage collection issue. If we ignore this we can easily see where the MPI_Send function is called. This is the place where mpiJava uses the underlying MPI implementation to send the data.

This approach has a number of distinct advantages and disadvantages. Firstly, by working on top of a complete MPI implementation, it is very easy to provide a complete version of the MPI Standard for Java. Also, because of the reliance on the native version of the library, any improvements made will then trickle up into mpiJava. However, by having to directly interface with the native C bindings Java loses all of its portability. Each machine will require the library to be recompiled and there may be incompatibilities between different versions. mpiJava also relies on an existing MPI implementation being available, which may not be the case on all types of system.

more information.
/**
   * Class: mpi_Comm
   * Method: send
   * Signature: (Ljava/lang/Object;IIILmpi/Datatype;II)
   */

JNIEXPORT void JNICALL Java_mpi_Comm_send(JNIEnv *env, jobject jthis,
   jobject buf, jint offset, jint count, jobject type, jint dest,
   jint tag)
{
  MPI_Comm mpi_comm =
      (MPI_Comm)((*env)->GetLongField(env,jthis,CommhandleID));
  MPI_Datatype mpi_type =
      (MPI_Datatype)((*env)->GetLongField(env,type,
         DatatypehandleID));
  int baseType = (*env)->GetIntField(env,type,DatatypebaseTypeID);

  void *bufptr;

  #ifdef GC_DOES_PINNING
  void *bufbase;

  clearFreeList(env);

  bufptr = getBufPtr(&bufbase, env, buf, baseType, offset);
  MPI_Send(bufptr, count, mpi_type, dest, tag, mpi_comm);
  releaseBufPtr(env, buf, bufbase, baseType);
  #else
  int size;

  clearFreeList(env);

  bufptr = getMPIBuf(&size, env, buf, offset,
      count, mpi_type, mpi_comm, baseType);

  MPI_Send(bufptr, size, MPI_BYTE, dest, tag, mpi_comm);
  releaseMPIReadBuf(bufptr);
  #endif /* GC_DOES_PINNING */
}

Figure 2.3: Native send method - mpi_Comm.c
2.2.2 MPJ Express

MPJ Express takes a different approach and provides a complete implementation of the Standard using only Java; much as the latteMPI hopes to achieve.

The Java-only approach allows MPJ Express to take advantage of the object-oriented style of Java to produce a well designed version of the MPI Standard. Java’s exception system allows errors to be handled without resorting to the checking of return codes. However, the conformance to Java’s style and syntax requires some compromise in how the functions are used. Pointers cannot be used, like they are in C, so it is difficult for functions to return multiple values in this way. All primitive types are passed by value, so these values would have to be returned in order to get them back into the user program.

Like MPICH, MPJ provides different layers of software. The MPJ API defines how the MPI Standard translates to Java objects and methods. At a lower level (the \texttt{mpjdev} package), the user can select between either a pure Java approach to communication or take advantage of a proprietary network device, depending on the hardware they have available.

The two main approaches to developing an MPI implementation in Java both have distinct advantages and disadvantages. latteMPI will be an implementation written solely in Java to take full advantage of its platform independent nature.

2.3 Applications of the Message Passing Interface

MPI has become ubiquitous in scientific computing, enabling a wide variety of applications to be written solving numerous scientific problems. There are a number of projects building on MPI to provide additional functionality. This section provides a look at these projects, providing context for latteMPI at a global level.

2.3.1 Scientific and Research Based Applications

Scientific applications are one of the fundamental areas for the application of message-passing. As we have seen, the message-passing model is a simple way of describing parallel computation. Parallelising computation can provide performance increases that are particularly important for scientific computing.
This section looks at two projects that use MPI and link it to other areas of research in distributed computing. Grid computing is similar to high-performance computing with clusters; it harnesses the power of multiple computational resources to run jobs. The key difference that grid computing provides is that these resources can be geographically separated. One could say that Grid computing is to clusters what the Internet is to local area networks.

Now that researchers have access to these computational resources, no matter where they are in the world, it is important that they can harness these resources with appropriate programs. One way to do this is to use MPI in a grid environment. The MPICH-G2 project builds on the MPICH implementation of the Standard, but provides functionality for handling issues that grid computing introduces [22]. The main issue with grids is heterogeneous resources (resources with differing hardware or operating system).

By adding extra methods to deal with heterogeneity, MPICH-G2 allows uses to program using the well established MPI methods whilst still taking advantage of new resources.

Apart from heterogeneity of nodes, another aspect of grid computing is node volatility. When using many resources from various locations, it is likely that resources will fail. MPI does not typically account for these failures, so the MPICH-V project builds on MPICH and adds features to ensure the node volatility is handled correctly [9].

Nodes disconnecting is typical in a grid system, since it is made up of many nodes from around the world. By providing features to checkpoint and restart MPI programs, users can be confident that they can run programs using grid resources without needing to worry about any volatile resources they might be running on.

### 2.3.2 The Gordon Bell Prize

The Gordon Bell Prize is awarded every year at the Supercomputing Conference, recognising achievements in supercomputing. The aim of the award is to track the progress of high-performance computing, as it improves every year. The Prize is given to the entry which has the highest performance in regards to floating point operations per second (FLOPS), but particular emphasis is placed upon applications that target scientific problems. The pervasive use of MPI in high-performance computing means it features regularly in the submissions of prize winners. The following sections present a brief discussion of
recent winners of the prize, highlighting both their use of MPI and the novel scales reached by these applications.


The application uses Qbox, a C++ implementation of this first-principles simulation method. Qbox relies on MPI for its message passing. The code was run on 65,000 processors and showed performance of 207 Teraflops ($10^{12}$ FLOPS). To provide some context of this scale, the recently installed supercomputer in the Centre for Scientific Complexity at the University of Warwick has 960 processors and a peak predicted performance of 11.5 Teraflops.

2008 The winning application in 2008 was a code that performed simulations of high temperature superconductivity [1]. Superconductors are materials that conduct electricity with zero resistance below a certain temperature. This research has profound implications for the electric grid, because electricity is currently transmitted at hundreds of kilo-volts to avoid any energy losses.

The simulation uses multiple layers or parallelisation, the highest level parallelisation used MPI for distributing data across nodes. OpenMP [7] is a tool used to provide thread level parallelism, where multiple threads of a process are run concurrently on a node. This was used by the application to obtain a performance of 400 Teraflops.

2010 The winning application in 2010 is presented in the paper Petascale direct numerical simulation of blood flow on 200K cores and heterogeneous architectures [33]. The application simulates blood flow in order to research microcirculation (blood flow in small capillaries). The blood flow is modelled using a mixture of plasma and red blood cells, and the largest run of this simulation used 200 million cells, compared to a previous maximum of 10,000 cells. Whilst this may sound like a lot of cells, it is only simulating around 50 microlitres of blood, approximately 50 ‘drops’.

The simulation can be run on both conventional CPUs as well as CPU/GPU machines, which use a combination of a CPU and a Graphics Processing Unit (GPU) to perform the computation. The simulation ran at 0.7 Petaflops ($10^{15}$ FLOPS) on 200,000 processors.
Every year, the winners of the Gordon Bell prize show significant advancement in the performance of real scientific codes. We can see here the leap from a peak performance of 207 Teraflops in 2006 to 0.7 Petaflops (700 Teraflops) in 2010. This kind of continued improvement shows the impact that scientific computing has on a wide range of scientific research. Also highlighted is the prevalence of MPI in high-performance computing. All three prizewinning codes discussed here have used MPI for at least part of their parallelisation.
CHAPTER 3
SYSTEM REQUIREMENTS

This chapter details the specification of the latteMPI project, as well as the corresponding requirements. The requirements are broken down into both functional and non-functional components. Section 3.4 identifies the limitations and the scope of the project. Section 3.5 presents a brief overview of the legal, social, ethical and professional issues surrounding the project, concentrating on the issues most applicable to a project like latteMPI.

3.1 Specification

The project will create a pure Java implementation of a subset of the most common MPI operations. This one sentence from the Specification (see Appendix A) encapsulates the requirements of the system. This will involve providing the MPI interface [27] to Java programs and implementing the routines necessary to perform point-to-point communications and collective operations. The project will also be able to initialise and finalise jobs using the semantics defined in the MPI Specification, enabling integration with existing job schedulers.

3.2 Functional Requirements

The requirements detailed here provide an objective overview of what the project needs to accomplish. The concrete goals enable the project evaluation to be precise, and will provide a means of justifying later design decisions.
1. **Provide a usable subset of the MPI Standard**

   This subset will need to include enough of the MPI Standard to enable useful computation. Breaking this requirement into some more concrete sub-points:

   (a) Management functionality, to start and end programs correctly.

   (b) Point-to-point communication routines. These are the simple send and receive routines that enable the most basic form of message passing.

   (c) Collective communication routines involve all processes and allow more complex tasks to be expressed with only one operation.

2. **Implement the subset according to the MPI Standard**

   It is important that the project implements its features to be consistent with the MPI Standard. This means:

   (a) The subset of operations satisfying the first requirement are implemented such that their semantics are equivalent to the semantics of the function defined in the MPI Standard.

   (b) All functions operate correctly regardless of the number of processors used for execution.

3. **Provide an implementation written in Java**

   Many MPI implementations exist for the standard languages, C and Fortran. Providing an implementation in Java means that the MPI functions will be provided ready to use by a Java program.

### 3.3 Non-Functional Requirements

The non-functional requirements of the project are less essential features that cannot be measured quantitatively. They encapsulate desirable features that will improve the usefulness of the final implementation.

1. **Acceptable performance**

   The project must be fast enough to perform meaningful computation in times that are comparable to existing C and Fortran implementations.
2. Extensible component architecture

Parts of the system must be able to be swapped out for new components that are tuned to particular hardware or features. For example, replacing the networking portion of the code in order to support a new network layer such as fibre optics. Whilst this could be achieved just by editing code, having well-defined components will make this process a lot easier.

3. Testable

Making the implementation testable will make it easy to show that the provided operations match up with the matching operations in the MPI Standard semantically.

4. Documentation

By documenting the project adequately, further work on latteMPI will not be hindered. New users or developers will be able to easily understand both the code and the design philosophy used.

5. Coding Standards

Consistent coding standards means keeping various aspects of the project consistent. Items such as file names, comments, spacing, and project layout, will need to be sensible and consistent. This will make the project easier to maintain or fix.

3.4 Scope and Limitations

The scope of the project is constrained to an acceptable level by choosing to implement a subset of all the operations defined in the MPI Standard. Whilst this constraining of the scope may be seen as a limitation, implementing the functions described in the requirements above presents a challenge that can be completed in the time frame allocated but still allows the completed implementation to run complex codes. Nevertheless, the restricted operation set may prove to have consequences during performance testing.

Many MPI implementations are produced by large teams of highly experienced engineers, so it would be foolish to expect my project to be able to compete with their implementations. This project will focus on a smaller subsection of the standard and aim to integrate well with the systems on offer at the Department of Computer Science at the University of Warwick.
3.5 Legal, Social, Ethical, and Professional Issues

3.5.1 Legal Issues

One of the most pressing legal issues for any software project is licensing. Many projects, including latteMPI, rely on code provided by others, and it is important to ensure that any licenses on these external code fragments are adhered to.

The latteMPI project uses four external pieces of software: jsch, jopt-simple, junit, and easymock. Each of these projects uses a separate license, but luckily all of these licenses are open source.

**jsch** The jsch library uses the BSD License [28], which originates the Unix-like operating system BSD.

**jopt-simple** The jopt-simple library uses the MIT License [29], which is a free software license that originated with Massachusetts Institute of Technology. In summary, the license says that any software releases under this license can be used freely, provided the license is included with any derivative works.

**junit** The junit code is released under the Common Public License [12], an open-source license published by IBM. The requirements of this license are similar to other open source licenses, in that the derivative work must include the original license. The Common Public License has now been superseded by the Eclipse Foundation’s Eclipse Public License, which is used by the popular Eclipse development environment.

**junit** The easymock library uses the Apache 2 License [3], which again states that derivative work must include the original license. The Apache 2 license was created by the Apache Software Foundation, a foundation that provides support for a large family of open-source projects [4], including Maven, the build system used by latteMPI.

The common theme to all these licenses is that derivative works are tightly controlled, generally to ensure that projects building on open-source software are released themselves under an open-source license. However, latteMPI does not use code from any of these projects, itself, it merely relies on the functions
they are designed to provide. All source code contained in latteMPI was written exclusively for the project.

3.5.2 Social Issues

Whilst the links between scientific computing and society may seem tenuous, computers have been an enabling technology for a number of projects with a positive impact on society. Projects like BOINC [40] and Folding@home [32] use computer power for research into a large number of scientific problems whose solutions would be of great benefit.

The Folding@home project uses the idle time of volunteers’ computers to run simulations of protein folding. Protein folding is a key biological mechanism, and incorrect folding of proteins is thought to be a component of many diseases. Understanding these foldings allows scientists to study diseases, with the end goal of preventing or curing them.

The BOINC project has a more general goal of harnessing idle computer power and enabling easy access to it. As with Folding@home, many of the uses for this power are medical, but the power is available to other disciplines, such as physics or climatology.

Whilst the latteMPI project is unlikely to be used for projects such as these, research in scientific and high performance computing fields, of which latteMPI is a part, continues to improve society with very visible results.

3.5.3 Ethical Issues

Scientific computing is often applied to domains that some may find ethically wrong. Examining the TOP500 [39] list of supercomputers highlights the amount of military and government backed computing projects that exist. Scientific computing is one of the key targets for latteMPI and any ethical issues surrounding these research projects will apply to latteMPI by proxy.

Regardless of the end use someone may find for the software, it is important to focus on the fact that this project was conducted primarily from a research and software development point of view. The goal of latteMPI is to provide a pure Java implementation of a subset of the MPI Standard, not to enable unethical computational research.

Regardless of the ethical soundness of the programs run using latteMPI, the author of latteMPI should not be held responsible for these anymore than a car manufacturer is responsible for speeding.
3.5.4 Professional Issues

The latteMPI project describes itself as a Java implementation of a subset of the MPI Standard, so adherence to this Standard is particularly important. The function of any MPI implementation is to provide a layer upon which end users can build, writing their own programs using the standard MPI functions.

The key distinction between an MPI implementation and other kinds of software is where the project will sit in terms of the whole system. An application built directly for users may well be built to looser standards, and many applications like this may contain quirks that are easy to avoid once users learn about them. An MPI implementation sits underneath another layer of code, the users program. This means that the application programmer will be relying on latteMPI to provide functions as they are defined in the MPI Standard.

The latteMPI project must provide the functions as they are described, in order to enable use by others, so that confidence that programmers naturally place in software libraries is not abused.
This chapter presents the design phase of the project. A narrative of the development of the design runs through the chapter, split into two sections, job startup, and message passing. Finally a summary of the design is presented.

4.1 System Design Overview

Figure 4.1 shows the system at an abstract level, presenting the various software layers. At the top of the diagram is the user program, at the highest level of abstraction. The user program will interact with classes in the package uk.ac.warwick.java.mpi in order to access the MPI functionality.

The classes in the mpi package will be supported by more classes in uk.ac.warwick.java.mpi.system and uk.ac.warwick.java.mpi.system.util. The system and util classes will provide the lower level operations in order to implement the MPI functions described in Section 4.3.1.

The most primitive layer of the design is the NetDevice, which is an interface in the system package. This interface provides the basic network transport functions upon which the rest of the implementation can build. Since NetDevice is an interface, it is easy to provide a different concrete implementation that can take advantage of particular hardware – Java over Infiniband, for example – yet still provide a “simple” implementation using Java’s basic TCP features.

The design takes advantage of the orthogonal nature [16] of the MPI Standard, as well as Java’s object-oriented design.
4.2 Startup

Initialising the environment for message passing presents a particular challenge for any implementation of MPI, but particularly for Java. Startup must be handled in a clean fashion, with ways to monitor the processes that have been started.

A Java application should be cross-platform, with little reliance on any operating system specific tools. This presents a challenge to job startup, since it is very system dependent. A Java class will need to be used to start the remote processes, to avoid adding any unnecessary dependencies to the project and limiting its portability.

Startup refers to everything that needs to happen in order for the users program to run correctly, as defined by the MPI Standard. The core of this is connecting to the remote machines and starting the correct number of remote processes, which corresponds to however many the user has requested. Prior to launching the users program on these remote machines however, a number of other steps will need to be completed. The environment on each machine needs to be set up, that is to say that any relevant variables are set correctly prior to starting the users program. This will include setting variables in the
The startup process in latteMPI will be accomplished by using a Java class to connect to each machine and run another Java program. This remote Java program will be responsible for setting up the environment and then launching the users program. The class will connect to the remote machines using ssh, and hostnames can be accessed from the PBS hosts file, so that they do not need to be manually entered by the user. These startup classes will be located in the system package, part of the lowest level of the latteMPI classes.

4.3 Message Passing

The main part of the project is providing the operations to the user program. The message passing operations will build upon the environment established by the startup routines and provide the vital parts of the MPI Standard that will be used by application programs.

4.3.1 Interface Design

The interface to be provided to the user needs to sit on top of the design of the underlying layers of the system. The system design provides a number of well defined classes that provide the MPI interface to the user. The MPI interface is the functions and constants defined in the MPI Standard, as they apply to Java and the object-oriented design of the system.

The MPI Standard provides a large number of operations ranging from the basic send and receive, to complex collective operations involving multiple groups of processes. Constraining the scope of the project is important, as a complete MPI implementation would take many months of work.

In [35], Skjellum presents what he describes as a “usable subset of MPI operations.” These operations provide the basic MPI functionality, so the implementation of these methods will be the primary goal of the project. The methods are presented in Table 4.1. Also presented in [35] are a small number of non-blocking methods; detailed in Table 4.2\(^1\).

\[^1\]These methods are presented with the naming convention used in the implementation, as discussed in Section 4.3.1
4.3 Basic functions:

- `Mpi.Init` Initialize MPI
- `Comm.Size` Number of processes
- `Comm.Rank` Find our process rank
- `Comm.Send` Blocking send
- `Comm.Recv` Blocking receive
- `Mpi.Finalize` Terminate MPI

<table>
<thead>
<tr>
<th>Collective communication:</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Comm.Bcast</code></td>
<td>All-to-one broadcast</td>
</tr>
<tr>
<td><code>Comm.Reduce</code></td>
<td>All-to-one reduction</td>
</tr>
<tr>
<td><code>Comm.AllReduce</code></td>
<td>All-to-all reduction</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Timing support:</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Mpi.Wtime</code></td>
<td>Relative time</td>
</tr>
<tr>
<td><code>Mpi.Wtick</code></td>
<td>Clock resolution</td>
</tr>
</tbody>
</table>

Table 4.1: Minimal subset of MPI operations

<table>
<thead>
<tr>
<th>Non-blocking communication:</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Comm.Isend</code></td>
<td>Non-blocking send</td>
</tr>
<tr>
<td><code>Comm.Irecv</code></td>
<td>Non-blocking receive</td>
</tr>
<tr>
<td><code>Comm.Wait</code></td>
<td>Wait for a message</td>
</tr>
</tbody>
</table>

Table 4.2: Minimal subset of non-blocking MPI operations

### Interface Naming Convention

The MPI Standard defines bindings (method and constant names and parameters) for both C and Fortran. These bindings do not map one-to-one into a neat Java interface, so some interpretation is required in order to end up with a usable set of bindings.

Luckily the MPI Standard specifies C++ bindings. Java is a language which has some similarities to C++, which evolved from C. It is natural to begin by looking at the C bindings.

The C bindings contain methods such as `MPI_Send()` and `MPI_Recv()`, and method names of this format would look out of place in a Java program. The C bindings also rely on passing parameters by reference, with the return value of the method being its success or failure; again, Java handles failure differently (with exceptions) and success can be assumed.

The C++ bindings are designed in such a way as to take advantage of the object-oriented nature of C++, so they are naturally a good fit for a Java implementation. C++ bindings have a form similar to `Comm::Send()`. This form would correspond to the Java style binding `Comm.Send()`, which looks natu-
ral to any Java programmer. Despite the C++ bindings being deprecated in version 2.2 of the MPI Standard it was decided that similar naming conventions would work well, as they have a natural affinity to the Java style of naming and also fit well within an object-oriented implementation design.

4.3.2 Constraining Datatypes

In the MPI Standard, users specify the memory location of the first element to be sent – using pointers like $\&a$ – and the number of elements to send. This works well for the C and Fortran since they allow explicit access to areas of memory. The MPI Standard also defines datatypes, which act as descriptions of the layout of data. These datatypes can be used for sending things like array columns or collections of multiple variables. Again, datatypes make use of the memory access traits of C and Fortran.

Java does not provide the kind of low level access needed to send data in this way, so data must be passed to the functions in a different way. In Java, primitive types (int, long, float, double, boolean) are passed by value and objects are passed by reference. You can only update the object pointed to by the reference; the reference itself cannot be modified in the same way as a C pointer.

The typical way to accommodate many different types of data in a function is to accept a parameter of type Object, which is the parent class that all other classes extend. While this is a nice way to view the data, it is a little too complex a solution to use when the majority of the time only numbers are being transferred.

The following points helped shape the decision about which objects would be supported by the project.

- Parallel applications generally deal with numeric problems, performing CPU intensive calculations before sending their results to other process for further work. This means that numeric data is the most common datatype to be send by MPI functions.

- When using Java’s objects, especially with a view to transmitting them across the network, a lot of overhead is introduced. All objects require creation, as well as deletion by the garbage collector. If you send an object across the network, then a lot of additional information about the object is sent by Java in order to ensure type-safety. This is a relevant
consideration in a high-performance system where speed is particularly important.

- As alluded to above, objects carry a significant size overhead, that is an important factor during network communication. Primitive types can be stored as bytes representing the data without any additional information required to describe the object. This results in a minimal amount of data being transmitted across the network.

These features all point to using the primitive numeric types as parameters. This is a viable choice, but limits data transfer to one item at a time. In order to overcome this primitive arrays will be used. Although arrays are objects, they can be transferred without the additional overhead normally required by objects since the primitive data they contain can be removed and sent, and then stored in another array at the remote process.

**Datatypes** The MPI functions in this project will support the numeric datatypes \texttt{int[]}, \texttt{long[]}, \texttt{float[]} and \texttt{double[]}.

### 4.3.3 Operations for \texttt{Reduce} and \texttt{AllReduce}

The reduction methods, \texttt{Reduce} and \texttt{AllReduce}, take a parameter describing the operation to perform on the data they collect from each process. The MPI Standard defines a range of operations that are permitted, ranging from simple arithmetic to complex user-defined operations. In order to constrain this functionality appropriately, only a small set of operations will be implemented.

Since latteMPI restricts the data types to numeric primitives, it makes sense to only support arithmetic operations. The chosen operations are ubiquitous in parallel programs, so will enable a large number of problems to be solved with latteMPI.

**Operations** The operation parameters for the reduction operations are: \texttt{sum}, \texttt{prod}, \texttt{max}, \texttt{min}, \texttt{maxloc}, and \texttt{minloc}. The first four operations work as you may expect, performing the named operation on the data. The last two operations perform the operation, and will return its result and well as the location of the maximum or minimum value respectively.
4.3.4 Additional Operations

Once implementation of the main communication operations was complete, small test codes were ran in order to exercise the system. As these codes were written it became clear that a number of functions would be useful when writing real-world programs.

**Barrier**

The first of the additional operations is barrier, a collective communication operation that acts as a synchronization point for all processes. Synchronization is useful in large parallel programs for separating different parts of the algorithm, especially if they have causal dependencies.

latteMPI will provide a Barrier operation, that will not complete on any process until all processes have entered.

**Processor Name**

A slightly more trivial addition is a utility method `getProcessorName`. This method will return the hostname of the machine running the local process. Whilst not a particularly complex method, it is useful for output purposes.

4.4 Design Summary

As is typical with most object oriented systems, the final design consisted of a number of classes and some information on how they will interact. Figure 4.2 presents these principal classes and shows the relationships between them. This class diagram refines the system design into a more concrete form, allowing implementation to begin. The final design encompasses the original specification, as well as the more detailed updates made throughout the course of the project.

The mpi package contains all the classes that are accessible to the user. These classes will contain the MPI message-passing methods and utility functions. The system package will contain classes providing lower level implementation. The MpiStarter class will be responsible for starting the required number of processes on the correct machines. The MpiProcess class will encapsulate this idea of a process, allowing the remote invocations of the program to be managed.
Figure 4.2: Principal classes of the system.
The `Processor` class represents a machine running a copy of the program and provides the MPI methods all the information they will need at runtime. The `NetDevice` encapsulates the lowest level of network communication.

The final package is the `op` package, and contains implementations of the six operations that can be used with the reduction methods, `Reduce` and `AllReduce`. 
This chapter charts the course of the implementation of the project, documenting both the work carried out and the product of that work.

In Chapter 3 the requirements of the project were presented. In order to satisfy the functional requirements, the implementation was broken down into 3 tasks:

- Implement initialisation routines to set up the runtime environment.
- Implement the network routines for data transfer between processes.
- Implement communication routines used by MPI operations.

The initialisation and network routines will provide a foundation on which the MPI communication routines can be built, and hence provide a usable subset of the MPI Standard.

The implementation of the MPI Standard can be split into two main components: starting up the jobs on distributed nodes, and then allowing the processes to send messages to each other.

### 5.1 Startup

Naturally, setting the environment up so that communication between nodes can be established is the first thing that should be accomplished.

Since the cluster to be used as a test environment is running the PBS job scheduler, the tool `pbsdsh` was researched. This distributes the task to multiple nodes using the PBS system. Relying on PBS however would have made
the system less portable, and particularly unsuitable for use on a single machine. This path was abandoned and alternatives were considered.

Java has some difficulty in accomplishing tasks such as these. Starting processes on a remote computer is simple, if you have access to normal system commands. Java’s platform independent nature removes these features making further work necessary to interact with operating system level tools.

For securely executing a remote command, on what may likely be a Linux node, ssh seemed like the perfect choice. The ssh tool allows remote login to machines, providing shell access to the remote machine. The processes are started using an sshexec which connects to the machine and runs the specified program, before closing the connection once the remote program terminates. There are a number of open source libraries that implement ssh available for Java, so it was simple to incorporate into the project.

Coincidentally, the cluster enables users to log into the nodes they have been assigned for a job by using an ssh key - this means we can ssh from the root node to start the remote processes without requiring a password or any form of user interaction.

The most difficult part of implementing the ssh commands in Java was finding a suitable library. The latteMPI project uses Jsch [20], a pure Java implementation of the SSH2 standard with a BSD-style license.
The ssh program is a good choice because of its portability. Implementations of ssh are available for the majority of computing platforms and are not confined just to the realm of enterprise or research systems. Using ssh decouples latteMPI from any particular job scheduler, and means it can just as easily be run on a laptop as a cluster.

Let us now proceed to the initialisation of the system. MpiStarter is the class that is run by default when the latteMPI JAR file is run. This takes a list of parameters: the (optional) file containing host names, the number of processes to create, and the name of the program to run. MpiStarter then creates a number of MpiProcessSshExec objects, which are responsible for connecting to the remote machines and executing the MpiRemoteProcess class. The MpiRemoteProcess class is then responsible for initialising the network communications and classes needed by the main Mpi class, before finally loading the user class using reflection.

Reflection  Reflection is how Java loads classes that have already been compiled, but are not known or present at compile time. This is essential for latteMPI, as it allows programs to be run without recompilation on the target machine. Due to the important role reflection plays in the system, we will take a more detailed look at this process.

The definition of reflection is a program that can modify itself or its behaviour at runtime. Loading pre-compiled classes is only a small portion of this, but is the most important for us to investigate.

Loading a class is relatively simple. The name can be provided dynamically and the Class.forName() method used to get an object that represents this class. Once this object has been created, it can be started with an invoke method. After this, the class will run exactly as if it had been run manually by the user. Once the class has completed its execution, the control returns to the calling method. This means that upon completion of the user program, latteMPI can still perform the necessary cleanup before closing the JVM.

Figure 5.1 presents a simplified overview of the initialisation of the system, with both the local and remote process being shown. It is important to note that MpiRemoteProcess is just the name of the class that initialises the MPI objects and loads the user program - this class was named before deciding that it would also be used on the local machine (the root machine).

By using the same code to create both remote and local processes, the sys-
tem has been simplified. What enables this is the fact that these connections are established using ssh, and it is perfectly acceptable to connect to a local machine via this protocol.

Figure 5.2 shows the objects created during the initialisation of the system. The Mpi class contains a static constant COMM_WORLD which is created before loading the user class. COMM_WORLD models a communicator, which is described by the MPI Standard as a group of processors. Each processor contains information such as the hostname and port it is listening on, so that this instance of the system can transmit data to that remote instance. One of the Processors is local, meaning it represents the current processor. This Processor has a NetDevice which controls all communication (discussed in Section 5.2).

5.2 Message Passing

Network communication lies at the heart of MPI; it is the way that data is transferred from one machine to another, albeit in a manner transparent to the user. Before discussing the abstract MPI communication routines, we will look at the low level detail of data transfer employed by the system.
As mentioned above (and alluded to in the Progress Report), the NetDevice interface provides the methods that handle all network communication. By using an interface, it is easy to extend the network portion of the system to support different communication links such as Infiniband. The next release of Java contains many new IO classes, provided by the package java.nio. Since latteMPI provides the NetDevice interface, it would be easy to write a new concrete implementation making use of these new IO classes, which may increase performance.

The SimpleNetDevice object uses the standard networking functions provided by Java, meaning it can be used on any platform with a full JVM. The object contains a server thread in a static inner class. This is started when the object is created and listens for connections. When a connection is made, a worker thread is spawned to receive and store the data. Received data is put into a Message object and stored in the MessageQueue. A Message is an object that encapsulates the concept of a message. It contains the actual data (stored as a byte array), as well as other envelope information that we will meet later. The MessageQueue is a hashmap that maps a \((source, tag)\) tuple to a linked list of matching messages. Messages are dequeued in the usual fashion, so that the first received message is the first delivered to the user program. To send data, the SimpleNetDevice simple connects to the remote process using its port number and transmits the data contained in the Message as byte array. Figure 5.3 shows this process diagrammatically, with the
transition of data indicated by the arrows. You will notice from this description that the communication uses an eager protocol. Data is sent and subsequently received without waiting for any acknowledgment.

The most basic routines are the blocking send and blocking receive, so that is where both the implementation and this explanation will begin.

### 5.2.1 Send

As mentioned in Chapter 4, latteMPI follows an object-oriented design strategy and the bindings will differ from the C and Fortran bindings specified in the MPI Standard.

Figure 5.4 presents the definition of Send for all the supported datatypes. As you can see, Send takes an array as well as a destination and tag value. The dest and tag parameters are used by MPI to identify the message and as we shall see shortly, to receive the correct message.

An ideal solution for the Send methods would have been Java’s generics, which allow methods to operate on a variety of types. However, generic methods are designed to operate on Java objects, not just primitive types. This means that the methods would have had to act on Integer[] arrays as opposed to int[] arrays, where Integer is the Java wrapper object for the primitive int type. Using the wrapper objects is not a natural way to program due to all the explicit conversions needed. Whilst Java does provide auto-unboxing (automatic conversion between primitive types and their wrapper objects), it is more complex than it seems and introduces a large amount of overhead. Therefore, a separate Send method will be provided for each type.

When Send is called, the data is packed into a Message object. This message contains the data, as well as the envelope information. This envelope information is:

- Source – the rank of the sending process.
5.2 MESSAGE PASSING

Figure 5.5: Message travelling through the system layers during the \texttt{Send} operation.

- Tag – the user-specified tag for the message.
- Type – an integer constant identifying the type of the message.
- Data - the data provided by the user, converted to a byte array.

This data is needed in order to correctly send the message and then deliver it to the user process running on the remote machine.

Once the Message has been created, the \texttt{Send} method passes it on to the object representing the group of Processors. The send method provided by the group is invisible to the user and operates on a slightly lower level. Here the message is passed to the send method of the \texttt{NetDevice}, where it is converted to a byte array and the actual transfer of data takes place. Figure 5.5 shows this transfer of data schematically.

In Section 5.2, the transfer of data was described as eager. This means that the data is transferred as soon as the method is called, regardless of whether a matching receive operation has been called on the remote process. As soon as the data has finished transferring the method will return control to the user program, regardless of whether the data has been delivered to the remote user program. Such synchronization is not required by the MPI Standard but it
is important to draw attention to it because of the ambiguous nature of the term *blocking*. In this case it simply means blocking until the data has been transferred, **not** until it has been received. The MPI Standard requires that a blocking method not return until the buffer is safe for re-use. In Java this is less of a problem that in C since memory is not accessed directly by the program. The implementation latteMPI uses simply blocks until the data has been sent, ensuring the buffer can be re-used as soon as possible.

### 5.2.2 Receive

The blocking receive operation is the natural partner of send. This relationship is at the heart of the message-passing paradigm and was mentioned in Chapter 1. Figure 5.6 shows the definitions for `Recv`.

The `Recv` method introduces another important object, the `Status` class. This class provides a wrapper around a number of values relating to the receiving of a message. The receive method fills in the elements of this class with the correct data and it becomes accessible to the user after the call has been completed. The data contained in a `Status` object is:

- **Source** – the source of the received message.
- **Tag** – the tag of the received message.
- **Error** – an integer code indicating whether the receive was successful.

At first glance the `Status` object might seem redundant, after all, we need to know the source and tag of the `Message` in order to call the `Recv` method in the first place. However, the `Status` does have a use.

The MPI Standard defines two constants `ANY_SOURCE` and `ANY_TAG`. These two constants allow the user to receive a `Message` regardless of its tag or source – which is useful if these values are either unimportant or unknown. latteMPI provides these constants as public final members of the `Mpi` class.
This means they can be referenced with the `Mpi.{ANY_SOURCE, ANY_TAG}` notation from the user program. If a message is received using one of these tags, then the programmer did not necessarily know exactly where the message would be coming from. The application can then use the `Status` object to find out the actual values of the source and the tag.

The `data` parameter of the `Recv` method needs to be an array of the same size and datatype as the data contained in the `Message` you are trying to receive. The received data is then copied into this array using a call to `System.arraycopy`, which is an optimised Java function for copying memory. An alternative to the array copying approach would have been to return the value from the method, however, since a `Status` also needs to be returned, updating the return parameters in this way (by reference) makes the most sense.

When the `Recv` method is called in a `Comm` object, the first method that is called is `Group.Recv`. This method returns a `Message` object which the `Comm.Recv` method will copy into the `data` array before populating the `Status` object and returning control to the user program. Inside the `Group.Recv` method, the `NetDevice.Receive` method is called. This method will search the `MessageQueue` by calling `get`. This call is wrapped in a `while` loop so that the process will block until the `Message` has been received.

Figure 5.7 shows the receive process schematically. Note that the network is not shown during this process, as it plays no part in the operation. All messages are received without intervention from the user program. This is due to the eager protocol discussed in Section 5.2.

**Message Queue**  The `MessageQueue` object was briefly mentioned in Section 5.2. It provides a data structure that holds `Messages` and enables their retrieval. The `MessageQueue` contains a hashmap – a data structure renowned for providing efficient storage and retrieval. The key of the hashmap is a `(source, tag)` tuple, the values of the hashmap are linked lists of `Messages`. The reason that the tuples link to linked lists rather than single `Messages` is purely for safety. If two messages arrived with the same source and tag, then the older would be overwritten with the newer. Clearly this is a huge error, so by queuing each message that arrives we can support many messages with same `(source, tag)` tuple correctly. It should be noted that by using a hashmap and a linked list, we can retain an average complexity approaching $O(1)$.

Continuing with our complexity analysis, it is important to disclose that the
Figure 5.7: Message travelling through the system layers during the `Recv` operation.

`MessageQueue` implementation will degrade in complexity if `ANY_SOURCE` or `ANY_TAG` are used. This is because the `get` method will need to iterate over the linked lists in the hashmap and return the first matching `Message`. If only `ANY_SOURCE` is used, then the complexity becomes $O(s)$ where $s$ is the number of sources that are in the hashmap. Respectively, the complexity increases to $O(t)$, where $t$ is the number of tags, when `ANY_TAG` is used. If both are used then the complexity returns to $O(1)$, because the very first `Message` in the `HashMap` can be returned.

### 5.2.3 Broadcast

Broadcast is the first collective communication operation that was implemented. A collective operation is one that is called in the same way by all processes. A broadcast sends the data from one process to all the others, and is shown in Figure 5.8. The process that will send the data is known as the `root` process. The `Bcast` operation provides the broadcast functionality included
5.2 MESSAGE PASSING

5.2. MESSAGE PASSING

The Bcast operation is different from the Send and Recv operations; they are primitives. Their nature requires them to be built on top of lower level components. With Bcast however, we are able to build on top of the simple Send and Recv operations in order to implement new functionality.

Figure 5.9 presents the definitions for the Bcast function for the four primitive datatypes that are supported by latteMPI. The methods only take two parameters: the data to be broadcast, and the rank of the root process.

The Bcast method is a lot more complex than the Send and Recv methods. Since we have already covered the low-level details of these two basic methods, we will look at Bcast from a higher level.

Let us first define a few variables that will ease the explanation of the concept: \(n\) is the number of processes in the system, \(data\) is the message to be sent, and \(r\) is the rank of the root process.

A naïve broadcast algorithm would send \(n - 1\) times on the process \(r\), with all other processes receiving the data. This algorithm would work correctly, but slowly. Imagine calling this operation when \(n = 100\). If the message to be sent was large then process 99 would be waiting a very long time to receive the data. The broadcast algorithm in latteMPI is more complex in order to obtain better performance and reduce idle time.
Algorithm 1 Pseudocode for the tree-broadcast algorithm.

```
for stage = 0 to \( \lceil \log_2(n) \rceil - 1 \) do
    if \( 2^{\text{stage}} \leq \text{rank} < 2^{\text{stage}+1} \) then
        receive from \( \text{rank} - 2^{\text{stage}} \)
    end if
    if \( \text{rank} < 2^{\text{stage}} \) then
        send to \( \text{rank} + 2^{\text{stage}} \)
    end if
end for
```

![Diagram of tree broadcast for 6 processes, rooted at node 0.](image)

Tree Broadcast  latteMPI uses a tree broadcast algorithm, which is a popular technique for improving the performance of broadcasts [37, 13, 19].

The tree broadcast algorithm used is described in [31] and is detailed in Algorithm 1. The algorithm broadcasts the message in a binomial tree starting at the root node. Each process runs the same loop, and will send or receive at the correct points.

A sample run of this algorithm would produce a tree like Figure 5.10. The algorithm is simplest to explain if we let the root node have rank \( r = 0 \). To use the algorithm for an arbitrary root, a function is used to calculate the relative root given the original rank of the root process. This ensures that the processes always rank in sequential order, cyclically if the root node \( \neq 0 \).

During the first iteration, the \( r \) will send to the process with rank \( 2^{\text{stage}} = 2^0 = 1 \). The process with rank 1 will enter the first if statement, and will receive from process \( 1 - 2^{\text{stage}} = 1 - 2^0 = 0 \). In the second iteration, both process 0 and process 1 will be sending the data. As you can see in Figure 5.10, 0 sends to 2, and 1 will send to 3. Both process 1 and 3 will have entered the first if statement.
and will receive the data.

Since the each node of the tree has $\leq 2$ children, the height of the tree will be $\log_2(n)$, where $n$ is the total number of processes. This height will reduce the runtime of the broadcast operation.

If the time taken to send a message is $s$ seconds, then the time taken to perform the naïve broadcast will be $n.s$ seconds. If the tree broadcast algorithm is used however, the runtime would be $\log_2(n).s$ seconds.

Now that we have grappled with this higher level of abstraction we can return to the lower level method calls and the trace of calls produced when Bcast is used.

All processes must call Bcast with the same arguments in order for it to complete successfully. On the processes that are not root, the data array must be of the correct size. The array does not have to be empty, but it will be overwritten during the call.

When Bcast is called, a number of variables are created. The rank of the process and the size of process group are stored as local variables for use in the loop. A Message object is created.

The loop described by Algorithm 1 is then entered. In the receive portion of the loop, the Message is fetched using the Recv method we met during the discussion of Comm.Recv(). The data contained in this message is copied to the array that was passed as a function to the parameter.

In the send portion of the loop, the Message to send must be created. On the root process, this is done by creating a Message using the array passed to the method. On all other processes, a Message will have already been received. If this is the case then the source is updated and the Message can then be sent to the correct process, using a call to Send. By recycling the received Message, we can avoid having to create a new one. Since the Message constructor requires all primitive data to be converted to bytes, this can be a costly operation when placed inside a loop.

Once the loop has terminated on each process, each process will now have a copy of the values from the data array on the root process.

5.2.4 Reduce

Reduce was the second collective operation to be implemented. A reduction is a collective operation where some function is applied to data from all the
CHAPTER 5. IMPLEMENTATION

5.2

Figure 5.11: A simple reduce operation.

```java
public void Reduce(int[] data, int root, MpiOp op)
public void Reduce(long[] data, int root, MpiOp op)
public void Reduce(float[] data, int root, MpiOp op)
public void Reduce(double[] data, int root, MpiOp op)
```

Figure 5.12: The `Reduce` operation.

processors and the result stored on the root processor. Figure 5.11 is a visual representation of this process. The method `Reduce` is defined in the MPI Standard and provides reduction functionality.

Reduce shares a similar skeleton to `Bcast` and can be viewed at a higher level than `Send` and `Recv`. Figure 5.12 presents the four `Reduce` functions provided by latteMPI.

The `Reduce` function involves first collecting $n$ copies of `data` on the process `root`. The `op` is then applied to all $n$ copies of the data, and the result copied into the `data` array on the root process.

**MpiOp**

Before we continue discussing the `Reduce` operation, it is important to understand the `op` parameter, which takes an object of type `MpiOp`.

`MpiOp` is an interface which defines the functions needed to apply this operation to the data collected during a reduction operation. It defines four `run` methods, which correspond to the four primitive datatypes supported by latteMPI. Each `run` method takes an `ArrayList` of arrays, and returns a single primitive array, which will contain the result of the operation.

latteMPI provides concrete operations for some common operators that can be used with `Reduce`. Table 5.1 details these operations.
### Table 5.1: Concrete MpiOps provided by latteMPI.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SUM</td>
<td>Sums the arrays on an element by element basis.</td>
</tr>
<tr>
<td>PROD</td>
<td>Multiplies the arrays on an element by element basis.</td>
</tr>
<tr>
<td>MAX</td>
<td>Finds the max of each of the corresponding elements.</td>
</tr>
<tr>
<td>MIN</td>
<td>Finds the min of each of the corresponding elements.</td>
</tr>
<tr>
<td>MAXLOC</td>
<td>Finds the max and the rank where it occurs.</td>
</tr>
<tr>
<td>MINLOC</td>
<td>Finds the min and the rank where it occurs.</td>
</tr>
</tbody>
</table>

Those familiar with design patterns will notice that this corresponds to the Command pattern in [14], sometimes called a functor. All this means is that the actual operation performed is not important to the Reduce method, as long as the run method of the operation conforms to the interface defined by MpiOp.

**Naive Reduce**  A naive reduction is the simplest way to implement the reduction functionality, but comes with increased cost in terms of time. The naive reduction algorithm works by first collecting all data on the root process, using \( n - 1 \) calls to Recv. Each copy of data is added to an ArrayList. Once all the data has been collected \( \text{op.run(a)} \) is called on the root process, with \( a \) being the array list of data. The result returned by run is then copied into the data array on the root process.

**Tree Reduce**  The naive reduction algorithm is trivial to implement on top of the Send and Recv methods already completed. However, the performance of this naive method can be improved by a more complex and efficient algorithm. The tree reduction algorithm works in the opposite way to the tree broadcast algorithm we met in Section 5.2.3.

latteMPI uses the tree reduction algorithm described by Algorithm 2. Figure 5.13 shows the tree that would be produced during a sample run of this algorithm.

Each line in Figure 5.13 corresponds to the sending and receiving of a message. At each node, the reduction is performed on all the data currently at that node, using whatever reduction operation was specified in the call to Reduce.

The processes are re-numbered relative to the parameter as they enter the Reduce function, in order for an arbitrary root value to be selected. Informally, at each stage of the algorithm half of the processes send their data, and the other half of the processes receive this data and perform the reduction. At the end of each stage, any process that performed a send in the previous iteration
Algorithm 2 Pseudocode for the tree-reduce algorithm.

```plaintext
data := input data
for stage = ⌈log₂(n)⌉ to 1 do
    if 2^{stage-1} ≤ rank < 2^{stage} then
        send data to rank − 2^{stage-1}
    end if
    if rank < 2^{stage-1} and rank + 2^{stage-1} < size then
        receive from rank + 2^{stage-1}
        perform reduction on received and stored values
        data := result of reduction
    end if
end for
```

Figure 5.13: Tree reduce for 8 processes, rooted at node 0.

can leave the function, as it is no longer required.

As with the tree broadcast, this tree has a height of log₂(n), giving a runtime bounded by $O(\log n)$, rather than $O(n)$ like the naïve algorithm.

Since reduction is a collective operation, each process must call the function with the same arguments. When `Reduce` is called, a flag is checked. This boolean variable simply controls an if statement that makes the reduction use either the tree or the naïve algorithm. This allows testing of both algorithms which is useful for performance measuring. Upon completion of the `Reduce` method, the process with rank root will hold the result of the reduction.

### 5.2.5 AllReduce

`AllReduce` is an all-to-all reduction, and the final collective operation that was implemented. An all-to-all reduction is a reduction where the result is dis-
5.2 MESSAGE PASSING

Figure 5.14: A simple allreduce operation.

```
public void AllReduce(int[] data, MpiOp op)
public void AllReduce(long[] data, MpiOp op)
public void AllReduce(float[] data, MpiOp op)
public void AllReduce(double[] data, MpiOp op)
```

Figure 5.15: The AllReduce operation.

tributed to all processes. Figure 5.14 shows this operation. AllReduce provides this functionality in MPI. Figure 5.15 shows the bindings that are provided by latteMPI for this operation.

Conceptually, the AllReduce method can be thought of as a Reduce followed by a Bcast operation. However, there are more efficient ways of providing this behaviour – in the same way that the tree broadcast was an improvement on the naive broadcast algorithm.

Since latteMPI already contains both Reduce and Bcast methods that use efficient algorithms, we can combine them to provide the behaviour defined by the AllReduce method.

This is exactly what the AllReduce method does. First the method calls Reduce with the provided data and reduction operation, using process 0 as the root. Once the reduction has completed, process 0 broadcasts its result to all the other processes. Since both methods use the efficient tree algorithms, the run time of Reduce is good.

Upon completion of the AllReduce method, every process will have a copy of the result obtained from performing the reduction operation on the data held by each process.
5.2.6 Barrier

The final communication method detailed in the design is Barrier, which was only added to the design once some programs were written using the latteMPI implementation. During the writing of these test codes it became obvious that a global synchronization point such as the one provided by Barrier is a necessity, both for benchmark style codes such as message ping-pong and for more complete codes like a parallel implementation of the Floyd-Warshall algorithm.

The Barrier method consists of one parameterless function, which must be called from all processors. Figure 5.16 provides the definition of the Barrier method.

The MPI Standard says that no process can leave the Barrier operation until all others have entered. In order to provide this behaviour, latteMPI uses an AllReduce. Since both the Reduce and Bcast methods uses a tree algorithm, they are not suitable for synchronization, even though they are collective.

Both tree algorithms allow processes to exit once they are no longer needed, this improves their performance but also prevents their use as synchronization points. By using the AllReduce, which is a combination of a Reduce and then a Bcast, latteMPI ensures that no process will leave before every other process has entered.

When a process enters Barrier, it will enter the call to AllReduce with an empty int array. The empty array is used so that only the envelope data will be transmitted for each message, ensuring that excess data transfer does not slow down the Barrier.

Inside this AllReduce as we saw previously, a Reduce then a Bcast is called. The key to ensuring the correct synchronization behaviour for the Barrier is this combination of operations. The Reduce cannot complete on the root process until all processes have entered it, since root is collecting the data from each process, albeit in a tree fashion. Even though a process may complete the Reduce before root, they cannot leave the Barrier because they will be waiting for root to send the result with Bcast.

This Barrier function was both simple and easy to implement and is a
useful addition to the set of operations provided by latteMPI.

5.3 Environment Management

Environment management is an important part of the MPI Standard, providing a way to access information that is relevant to parallel programs.

Many programs will use a root process, that is responsible for distributing and collecting results. The other processes will simply perform calculations and then return their results. In order to program in this fashion, the process must be able to access its own rank, in order to branch into the correct sequence of statements.

Init  The Init method is used to initialise the system. Variables and objects are created and initialised, ready for the user program to run.

Finalize The Finalize method ends the MPI environment. In Java, this corresponds to setting variables to point to null, so that the objects they were pointing too can be cleared by the garbage collector. Generally an MPI program will exit straight after the call to Finalize. However, since memory management can not be performed explicitly in Java, it is important to mark objects for garbage collection so that the user application will receive more memory for any additional processing it performs that does not require MPI.

Size  When the Size method is called on a communicator, it returns the number of processors in that communicator. Since the Processor objects are stored as an ArrayList latteMPI can simply call the Size method provided by this ArrayList.

Rank  The Rank method returns the rank of the calling process in the communicator it was called on. This is done by finding the index of the LocalProcessor in the corresponding ArrayList of processors.

Wtime  The Wtime method can be used for benchmarking. It returns the number of seconds since an arbitrary point in the past, using the System. getNanoTime method provided by Java. This is the most accurate timing method Java provides. However, since this provides the time in nano-seconds the result must be divided in order to return this time in seconds. To avoid
the use of a *magic number* in the latteMPI code, this value \((10^{-9})\) is stored in the `TO_SECONDS` constant.

**Wtick**  The **Wtick** method provides the resolution of the clock used for the **Wtime** method. In Java is is unsafe to rely on operating system specific elements like the system clock. Because of this, latteMPI calculates **Wtick** by subtracting the difference between 2 successive calls to `System.nanoTime()`. Since the `System.nanoTime()` method is implemented using the most accurate timer available to the system, the difference between two successive calls should be close to the resolution of the system clock.

**getProcessorName** latteMPI implements this method by calling the `getHostName` method provided by the `java.net` package. As described in the MPI Standard, this method only needs to return the hostname of the local processor, so this method is perfectly acceptable.

All of the environment management methods seemed to mainly map one-to-one onto existing methods provided by the Java API. This meant that these methods are reliable and were also very simple to implement.

### 5.4 Job Cleanup

The final step in the runtime life cycle of any MPI program is the job termination and cleanup. The implementation employed by latteMPI cleanly handles both successful job termination and forced termination, either by the job scheduler or the user.

Since each process was started in a separate thread on the root node, these threads persist on the root machine for the duration of the programs run, and are responsible for printing output fed back to them by the process executing over ssh.

Once the program has called the `Mpi.Finalize()` method, and has reached the end of its `main` method, the method call that started the program will return its exit code. The `MpiRemoteProcess` will send this code to the `MpiProcessSshExec` method, which will print the exit code on the root node. The `MpiProcessSshExec` method will then finish, and the thread containing it will stop running. Once all threads have returned, the job has finished successfully.
In the case of the job being terminated prematurely, either by the user or the PBS scheduler, the main MpiStarter will be the only process terminated, since all processes started by ssh are executing independently of that JVM instance. In order to ensure that no stray processes are left executing, a shutdown hook is added to the JVM by the MpiStarter class. The shutdown hook is run by the JVM before it closes, and simply tells each thread to terminate the remote process that it is responsible for. The threads are able to do this by sending a kill signal through the ssh connection.
CHAPTER 6

TESTING

This chapter presents the testing of the system from two different perspectives. The unit tests ensure that the smallest components of the system are functioning as expected. System testing ensures the entire project works as a cohesive program, relying on the correctness of the components that has been guaranteed by unit testing.

6.1 Unit Testing

Unit testing is at the core of latteMPI, with unit tests being designed for each function as they were implemented. Unit testing is a simple concept – a unit test is a small piece of code designed to test the interface to a class.

latteMPI uses the JUnit framework [21], which provides a set of classes and methods designed to facilitate unit testing of Java programs. The method in Figure 6.1 demonstrates some of the key features of unit testing with JUnit.

The @Test annotation indicates to the test-running code that this method should be run and treated as a test. Inside the method test data is created, here it is an ArrayList conforming to the interface provided my MpiOp which MaxLoc implements. The run method is then called on this list. We know that MaxLoc will return the maximum of each of the corresponding elements in the arrays. We can also see from the test data the location of these maximum elements. Because of this we can now use the assertArrayEquals statements to ensure that our beliefs about the behaviour of MaxLoc are correct. If an assert statement returns false when the test is being run, then the test will be
Figure 6.1: The testInt method – unit test.

marked as failed in the output file, and we know that there is a problem with
the method in question.

The object oriented nature of latteMPI means that some classes rely on ob-
jects from other classes. However, a key principle of unit testing is isolating
classes in order to test them, minimising any side effects. In order to maintain
isolation whilst still testing code relying on other objects, mock objects can be
created. A mock object is an object under full control of the developer that
takes the place of any other object during testing. Mock objects allow the pro-
grammer to specify what methods they should respond to and what values
or objects these methods should return. This ensures the class being tested
is only relying on the interface of this other object. A framework is needed
for constructing these mock objects; latteMPI uses the easymock library. The
easymock library provides methods for simple creation and control of mock
objects, that can fill in for any other class. At the end of a test, various prop-
erties of the mock object can be tested, ensuring that the test code made all the
method calls that were expected of it, and didn’t make any unexpected calls to
the object.

6.1.1 Running the Unit Tests

In keeping with the agile development style, unit tests were written in tandem
with the implementation of the project, enabling continuous testing of the soft-
ware. Upon completion of the project, the unit tests exercised 71 percent of all
code in the project. Since the focus of a unit test is so small it is impractical
to include every single test here, so Figure 6.2 contains the example output for
one test run of latteMPI.
In the output, you can see that each class is tested using the unit tests that are defined for it. After running the tests a summary is printed, these are the lines similar to:

Tests run: 26, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 1.742 sec

Once all tests have been run, a final summary is printed and the testing program completes. It is important to note that unless all unit tests pass, the project cannot be built. This ensures that the project cannot be accidentally used in an erroneous state, and serves as a useful way to catch any bugs in the implementation. The example output highlights one fantastic feature of unit testing. Since all the tests are so small and isolated, testing the entire project takes in the order of seconds. This means that the project can be tested after every single change, however minor, to ensure that nothing has broken.

### 6.2 System Testing

System testing focused on running various simple, yet complete, test programs that exercised the various MPI functions that latteMPI implements. These programs are detailed in Table 6.1 and included in the `test_progs` directory on the CD that accompanies this report.

Each program was run multiple times with varying node counts and the results examined by hand. If the correct result was obtained on every run of the program then this test program would be considered successfully run, and the functions it used tested.

<table>
<thead>
<tr>
<th>Program</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ping_pong_2p</td>
<td>Tests <code>Send</code> and <code>Recv</code> between two processes.</td>
</tr>
<tr>
<td>ping_pong_np</td>
<td>Pairs off ( n ) processes, and has each pair perform a ping-pong.</td>
</tr>
<tr>
<td>mpi_trap</td>
<td>Simple parallel program to estimate the area under a curve.</td>
</tr>
<tr>
<td>TestBcast</td>
<td>Uses <code>Bcast</code> to send a message to all processes.</td>
</tr>
<tr>
<td>TestReduce</td>
<td>Uses <code>Reduce</code> to perform a simple reduction of data.</td>
</tr>
<tr>
<td>TestAllReduce</td>
<td>Uses <code>Allreduce</code> to perform a reduction, with the result being stored on all processes.</td>
</tr>
</tbody>
</table>

Table 6.1: Programs used for system testing.

The first test programs used only single calls to each of the communication methods. The `Send` and `Recv` methods were tested by sending an array from one process to all others. Each receiving process would then print its array, and this output could then be checked against the array that was being sent.
Figure 6.2: Output of running unit tests on the latteMPI code.
The first program to perform any meaningful work was the mpi_trap program, a Java version of a simple parallel trapezoidal rule program from [31]. This still only uses the Send and Recv functions, but calculates the area under a curve by using the trapezoidal rule (summing a trapezoid approximation of the curve).

The final test programs exercised the Bcast, Reduce, and Allreduce methods. This was accomplished by performing the operations with known data, and printing the result from each process. The results were then checked against the expected values.

6.2.1 Performance Evaluation Programs

The programs used for performance evaluation provided the final part of the system testing. The three programs: ping-pong, Floyd-Warshall, and SWEEP3D, are explained in detail in Chapter 7. We do not need to discuss the programs to explain how they provided the final system test. The programs were run alongside existing, correct, C versions. Upon completion of each run, the results of the latteMPI Java version were then verified against the results from the C code. This final set of tests ensured the correctness of the latteMPI implementation.
CHAPTER 7

PERFORMANCE EVALUATION

This chapter presents the final accomplishment of the latteMPI project: the results of performance benchmarks and comparisons of the runtime of real-world codes.

Performance is important for any MPI implementation, and given Java’s poor reputation for speed, it is even more important here. Whilst expecting a language that runs on a VM to exceed the performance of a compiled language like C is unrealistic, it is reasonable to hope for performance that is acceptable.

The performance evaluation serves to differentiate latteMPI from other Java MPI implementations. Whilst they are more complete and have been developed by teams of experienced programmers, they have not been used for running any notable scientific codes. If, by porting a non-trivial scientific application to Java and running it with latteMPI, correct and timely results are produced then the impact of the project will be a lot larger than if it were merely a limited MPI implementation with a small collection of trivial test programs.

7.1 Experimental Methods

The performance tests were carried out on the deepthought cluster in the Department of Computer Science. The cluster consists of 42 nodes, each containing two 1.4GHz Intel Pentium III CPUs and 2GB of RAM. The cluster nodes are connected by Myrinet fibre optics and a fallback Gigabit Ethernet (GbE) network. Since Java is not configured to use the Myrinet network, the gigabit ethernet interconnect was used. The C programs were compiled and run using
Open MPI 1.2.7, and the Java programs were compiled and run on the Open JDK Server VM 1.6.0.

7.2 Ping-Pong

The simplest benchmark for any MPI implementation is ping-pong. The ping-pong benchmark measures the round trip time between two nodes. The roundtrip time is the time taken to send a message from one node to another, where it is received and sent back to the starting node. When comparing the results produced by latteMPI to the results of C implementations, the overhead introduced by Java will become clear.

In order to accurately judge the performance, message size was varied in factors of two from 8 bytes to 16 megabytes. 1000 round trips were performed for each message size, and the time averaged over these 1000 iterations.

Figure 7.1 presents the results of this benchmark graphically. As expected, the round-trip times of latteMPI were higher than the equivalent times from Open MPI. However, as the message size increases, time increase is fractionally less. By extrapolation, we can see that as the message size continues to increase, the performance difference will become smaller as a proportion of the time taken by Open MPI.

Figure 7.2 displays the results as two separate graphs, to highlight the times
7.2. PING-PONG

Figure 7.2: Results of the ping-pong benchmarks as two separate graphs.

(a) Message sizes 1 to 1M bytes. (b) Message sizes 1M to 16M bytes.

take for very small messages. Figure 7.2a uses a logarithmic scale on the y-axis, and this demonstrates the big difference between the three tests when the message sizes are small. For the smallest 8 byte message, latteMPI took 40 times longer than Open MPI using ethernet.

The graphs also include timings for running Open MPI using the Myrinet interconnect. These timings are a lot quicker and it is unfortunate that running Java over interconnects other than ethernet is non-trivial. Based on the data collected, running latteMPI over a faster interconnect would hopefully scale well, providing fast round trip times, with a slight overhead compared to Open MPI.

Converting data into bytes is a costly operation for Java. Unlike C, where memory locations can be accessed directly and bytes easily read, Java has a more complex memory structure. This means that any primitive data type needs to be manually converted (marshalled) to a byte representation prior to sending. Judd, Clement and Snell highlight the performance degradation introduced by marshalling in [26], paying particularly attention to marshalling in the context of Java and MPI.

Table 7.2 shows the time taken to convert a message to and from bytes to the primitive data type *double*. This highlights the overhead that Java introduces with this required data marshalling. A ping-pong with 16Mb of data takes approximately 10 seconds, however, 1.4 seconds of this is marshalling time. So, latteMPI only takes an additional 0.6 seconds of other overhead when compared to Open MPI (which does not need to marshall any data).
## Table 7.1: Table of results for the ping-pong benchmark. Sizes detailed are $2^n$ increments for brevity.

<table>
<thead>
<tr>
<th>Size (bytes)</th>
<th>latteMPI (s)</th>
<th>Open MPI (s)</th>
<th>Open MPI [Myrinet] (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>0.083575</td>
<td>0.002292</td>
<td>0.000026</td>
</tr>
<tr>
<td>16</td>
<td>0.142285</td>
<td>0.004002</td>
<td>0.000061</td>
</tr>
<tr>
<td>32</td>
<td>0.189853</td>
<td>0.005304</td>
<td>0.000088</td>
</tr>
<tr>
<td>64</td>
<td>0.257146</td>
<td>0.006605</td>
<td>0.000116</td>
</tr>
<tr>
<td>128</td>
<td>0.322369</td>
<td>0.007909</td>
<td>0.000147</td>
</tr>
<tr>
<td>256</td>
<td>0.413624</td>
<td>0.009733</td>
<td>0.000182</td>
</tr>
<tr>
<td>512</td>
<td>0.491018</td>
<td>0.011955</td>
<td>0.000225</td>
</tr>
<tr>
<td>1024</td>
<td>0.565206</td>
<td>0.013579</td>
<td>0.000283</td>
</tr>
<tr>
<td>2048</td>
<td>0.646325</td>
<td>0.015438</td>
<td>0.000376</td>
</tr>
<tr>
<td>4096</td>
<td>0.700224</td>
<td>0.017617</td>
<td>0.000509</td>
</tr>
<tr>
<td>8192</td>
<td>0.752316</td>
<td>0.021587</td>
<td>0.000727</td>
</tr>
<tr>
<td>16384</td>
<td>0.807629</td>
<td>0.026333</td>
<td>0.001100</td>
</tr>
<tr>
<td>32768</td>
<td>0.862727</td>
<td>0.034061</td>
<td>0.001741</td>
</tr>
<tr>
<td>65536</td>
<td>0.925088</td>
<td>0.050936</td>
<td>0.002962</td>
</tr>
<tr>
<td>131072</td>
<td>1.001116</td>
<td>0.078748</td>
<td>0.005211</td>
</tr>
<tr>
<td>262144</td>
<td>1.104071</td>
<td>0.128373</td>
<td>0.009669</td>
</tr>
<tr>
<td>524288</td>
<td>1.269652</td>
<td>0.222803</td>
<td>0.018253</td>
</tr>
<tr>
<td>1048576</td>
<td>1.550315</td>
<td>0.406413</td>
<td>0.034142</td>
</tr>
<tr>
<td>2097152</td>
<td>2.103316</td>
<td>0.768501</td>
<td>0.064806</td>
</tr>
<tr>
<td>4194304</td>
<td>3.171161</td>
<td>1.574489</td>
<td>0.125138</td>
</tr>
<tr>
<td>8388608</td>
<td>5.392509</td>
<td>3.456202</td>
<td>0.244175</td>
</tr>
<tr>
<td>16777216</td>
<td>10.137308</td>
<td>8.030370</td>
<td>0.480241</td>
</tr>
</tbody>
</table>
### Table 7.2: Time taken to convert doubles to/from bytes.

<table>
<thead>
<tr>
<th>Size (bytes)</th>
<th>Time to bytes (s)</th>
<th>Time from bytes (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>0.000007</td>
<td>0.000007</td>
</tr>
<tr>
<td>16</td>
<td>0.000014</td>
<td>0.000007</td>
</tr>
<tr>
<td>32</td>
<td>0.000012</td>
<td>0.000010</td>
</tr>
<tr>
<td>64</td>
<td>0.000017</td>
<td>0.000017</td>
</tr>
<tr>
<td>128</td>
<td>0.000030</td>
<td>0.000031</td>
</tr>
<tr>
<td>256</td>
<td>0.000057</td>
<td>0.000058</td>
</tr>
<tr>
<td>512</td>
<td>0.000109</td>
<td>0.000113</td>
</tr>
<tr>
<td>1024</td>
<td>0.000215</td>
<td>0.000223</td>
</tr>
<tr>
<td>2048</td>
<td>0.000457</td>
<td>0.000501</td>
</tr>
<tr>
<td>4096</td>
<td>0.000856</td>
<td>0.000888</td>
</tr>
<tr>
<td>8192</td>
<td>0.001769</td>
<td>0.001821</td>
</tr>
<tr>
<td>16384</td>
<td>0.003533</td>
<td>0.003626</td>
</tr>
<tr>
<td>32768</td>
<td>0.007025</td>
<td>0.007212</td>
</tr>
<tr>
<td>65536</td>
<td>0.014064</td>
<td>0.033754</td>
</tr>
<tr>
<td>131072</td>
<td>0.047664</td>
<td>0.017726</td>
</tr>
<tr>
<td>262144</td>
<td>0.013641</td>
<td>0.042217</td>
</tr>
<tr>
<td>524288</td>
<td>0.031259</td>
<td>0.010467</td>
</tr>
<tr>
<td>1048576</td>
<td>0.040688</td>
<td>0.022272</td>
</tr>
<tr>
<td>2097152</td>
<td>0.044658</td>
<td>0.039057</td>
</tr>
<tr>
<td>4194304</td>
<td>0.142863</td>
<td>0.084446</td>
</tr>
<tr>
<td>8388608</td>
<td>0.374391</td>
<td>0.399438</td>
</tr>
<tr>
<td>16777216</td>
<td>0.660431</td>
<td>0.633117</td>
</tr>
</tbody>
</table>
7.3 Floyd-Warshall

The first real program written for latteMPI was an implementation of the Floyd-Warshall algorithm. The all-pairs Floyd-Warshall algorithm finds the shortest paths between all pairs of vertices in a graph.

The implementation used represents the graph as a 2-D array, with the element \((i, j)\) corresponding to the cost of the edge \(i \rightarrow j\). This array is distributed amongst the processes as a series of rows. The processes iterate over the vertices, and at each iteration a column is exchanged containing the shortest paths that the current node knows about. This column is broadcast to the other processes who then use it to update the distances in their local portions of the array.

Whilst ping-pong is a very simple benchmark only measuring message passing latency, the Floyd-Warshall algorithm contains a mix of both computation and communication, making it an interesting second performance test.

The Java and C implementations of the algorithm use the MPI \(\text{Wtime}\) function to time the total wall time taken for the algorithm to complete.

7.3.1 Floyd-Warshall 1000 Node Graph

The first input size tested was a graph with 1000 nodes. Figure 7.3 contains the results of this benchmark. Table 7.3 contains the data used in the plotting of the graph.

<table>
<thead>
<tr>
<th>Number of processors</th>
<th>latteMPI (s)</th>
<th>Open MPI (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>340.925</td>
<td>222.552</td>
</tr>
<tr>
<td>2</td>
<td>322.264</td>
<td>144.788</td>
</tr>
<tr>
<td>3</td>
<td>211.536</td>
<td>110.587</td>
</tr>
<tr>
<td>4</td>
<td>238.930</td>
<td>147.985</td>
</tr>
<tr>
<td>5</td>
<td>215.717</td>
<td>59.011</td>
</tr>
<tr>
<td>6</td>
<td>191.942</td>
<td>74.977</td>
</tr>
<tr>
<td>9</td>
<td>194.229</td>
<td>30.716</td>
</tr>
</tbody>
</table>

Table 7.3: Table of runtimes for the Floyd-Warshall program.

When only 1 processor is used, Java is approximately 100 seconds slower than C. This overhead is purely computation time, as there is no message passing taking place with only one processor. When two processors are used, both implementations reduce their runtime. However, the overhead of Java increases to approximately 150 seconds.
7.3.2  Floyd-Warshall 5000 Node Graph

Extending the Floyd-Warshall benchmark to use a 5000 node input graph will provide more computational work for the processors. This will reduce the proportion of time that the processors are communicating, which is where latteMPI suffers most.

Figure 7.4 shows the performance of latteMPI and Open MPI with the 5000 node graph. Much like the 1000 node graph, both implementations experience speedup when more processors are added. The overhead of latteMPI is approximately in the range 100-300 seconds. The C implementation of Floyd-Warshall suffered from memory errors with more than 9 processors. At 10 processors the runtime of the latteMPI implementation spiked very significantly, yet interestingly the trend of the runtimes then decreased in a similar fashion to the first 9 runs, albeit around 500 seconds slower.

7.4  SWEEP3D

SWEEP3D is a benchmark code that solves a 3D transport problem [5]. This complex scientific application is a good opportunity to really test the limits of latteMPI in terms of its stability and speed.

The SWEEP3D algorithm needs an input file to specify the parameters for the problem. Figure 7.5 contains an example input file. The $x$ and $y$ parameters
are used to specify the number of processors in the decomposition. SWEEP3D expects the total number of processors used to be equal to $x \times y$. On the second line, the first three numbers (500 500 500 in Figure 7.5) denote the problem size in the $x$, $y$ and $z$ directions. Testing was done on five input sizes: $50^3$, $75^3$, $100^3$, $250^3$ and $500^3$. The SWEEP3D program records the runtime of the algorithm, and outputs it along with a solution when the program completes. Like the implementation of Floyd-Warshall, both C and Java use MPI’s $Wtime$ function in order to time the algorithm. This runtime value was used in the performance comparisons that follow.

### 7.4.1 SWEEP3D 215 Thousand Cell Problem

The smallest input was $50^3$, for a total of 215,000 cells. The problem was run on square numbers of nodes from 1 to 25, due to the problem being decomposed in the $x$ and $y$ directions. Figure 7.6 shows the results of the tests. The perfor-

![Graph of results for Floyd-Warshall 5000 node graph benchmark.](image)

Figure 7.4: Graph of results for Floyd-Warshall 5000 node graph benchmark.

Figure 7.5: Example input file for SWEEP3D
7.4. SWEEP3D

Performance of latteMPI is very competitive until the 5x5, 25 node, decomposition, where the runtime then starts to increase dramatically. This could be due to the larger number of message being sent, and the fact that only square numbers are used shows this rather suddenly.

7.4.2 SWEEP3D 420 Thousand Cell Problem

In the \(75^3\), 420,000 cell problem, the performance is similar to the \(50^3\) case. As is visible in Figure 7.7, the performance of latteMPI is satisfactory until 25 nodes. In decompositions of more than six nodes, the performance suffers heavily and runtime increases linearly, whilst Open MPI continues to reduce its runtime.

7.4.3 SWEEP3D Million Cell Problem

An input size of \(100^3\) was used next. The problem size is relatively small, but still results in a large amount of both communication and computation. In the 5x5 decomposition, there are 6400 global messages per iteration, and the algorithm performs a total of 12 iterations in each run.

Figure 7.8 contains the results of the benchmark of this problem. When only 1 processor is used, Java has an overhead of approximately 100 seconds, much like in the Floyd-Warshall benchmark. In the 2x2 decomposition, both implementations decrease in runtime, but the overhead of Java increases to
Figure 7.7: Graph of results for SWEEP3D benchmark using problem size $75^3$.

Figure 7.8: Graph of results for SWEEP3D benchmark using problem size $100^3$.

<table>
<thead>
<tr>
<th>Number of processors</th>
<th>latteMPI (s)</th>
<th>Open MPI (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1x1</td>
<td>328.652</td>
<td>225.154</td>
</tr>
<tr>
<td>2x2</td>
<td>200.902</td>
<td>50.120</td>
</tr>
<tr>
<td>3x3</td>
<td>209.172</td>
<td>23.419</td>
</tr>
<tr>
<td>4x4</td>
<td>195.092</td>
<td>17.210</td>
</tr>
<tr>
<td>5x5</td>
<td>223.127</td>
<td>9.034</td>
</tr>
</tbody>
</table>

Table 7.4: Table of runtimes for SWEEP3D on input size $100^3$. 
approximately 150 seconds. This is largely due to the added communication overhead. As the number of processors increases, the runtime improvements begin to cease, and on latteMPI the 5x5 decomposition takes longer than the 4x4. It is likely that this is due to the increased communication time, coming from the large number of messages, outweighing the reduction in computation gained by adding more processors. It is because of this that the next benchmark uses an increased problem size, in order to hopefully decrease the ratio of computation time.

7.4.4 SWEEP3D 15 Million Cell Problem

The 15,000,000 cell problem uses an input of \(250^3\). Here, the performance shown by latteMPI is extreme, as we can see in Figure 7.9. The 2x2 decomposition, using 4 processors, takes 3588 seconds, but the next decomposition, 3x3, takes only 575 seconds. Performance then closely matches the C implementation until 5x5 nodes. At the seemingly ‘magic’ number of 36 nodes, the runtime explodes and then continues to grow as more processors are added. For Open MPI the runtime continues to steadily decrease.
CHAPTER 7. PERFORMANCE EVALUATION

7.4.5 SWEEP3D 125 Million Cell Problem

The largest input size used was $500^3$, corresponding to 125,000,000 cells. The results of this test are shown in Figure 7.10. Such a large problem size really showed the weakness of latteMPI. The runtimes for 36, 49, and 64 nodes are vastly greater than the Open MPI runtimes measured. Again this may revolve around using 36 nodes, however, starting with this decomposition was essential in order to ensure the results were reached in a tractable time frame.

7.5 Conclusion

Unsurprisingly, latteMPI and Java have proven to be slower than C and Open MPI. However, the overhead is reasonable and there are likely to be particularly circumstances where latteMPI or another Java MPI implementation makes sense. The main problem size where latteMPI seems to excel is on node counts between 2 and 25. Once more nodes are added and the number of messages increases, the overheads introduced by Java start to add up and dominate the run times. If rapid deployment is needed, then the compile once and run anywhere nature of Java is a definite advantage. Setting up an MPI implementation like Open MPI can take a long time. In particular, compiling the library can take hours on slower systems. With Java, compilation of the library
is not needed. Simply copy the latteMPI .jar file into the same directory as your program and then use it like any other Java library. In situations such as these the overhead will be of much less concern, since the program could have already been running for 20 minutes before Open MPI has even been compiled.
This chapter presents the project management of latteMPI, as well as related issues. The project management is evaluated chronologically and the related issues of development process and tools are tackled at the end of the chapter.

8.1 Design Phase

The design phase of the project is arguably the most difficult. Conceptualising a large software system provided a real challenge and there is room for improvement in the work products of this phase.

In the Progress Report, I mentioned that I was eschewing the class diagram in favour of a more abstract system diagram (presented in Figure 4.1). Whilst this abstract version is more suitable for presenting the design ideas of the project, I have found the diagram to be less helpful.

A well designed class diagram would have provided me with a lot more information, and confidence that the code I was producing would fit well with the rest of the system. Software design and particularly object-oriented software design are complex topics, so the abstract approach I took was definitely the easier one. With more practice and less time constraints, I feel a more elegant and concise system would be well within my grasp.

As any computer scientist will know, programming can be a slow task, so I was keen to limit the time I spent on designing the system, in order to begin the implementation. In hindsight I think that more time designing would have allowed me to implement the project quicker, but it is only easy to think this
after having completed the project.

Management of the design phase went smoothly. Table 8.1 includes the tasks I allocated myself for Term 1 (including the whole design phase). Figure 8.2 (a small section of the complete timetable presented in Appendix A) shows the time allocated to complete the design phase was 6 weeks. I kept to this schedule and completed the design work on time. Had I attempted to produce a full detailed class diagram, I think that the design work would have overrun and affected the rest of the project. Given that without an implementation there would not be much of a project, I think simplifying the design phase was the correct decision.

### 8.2 Implementation Phase

The implementation phase of the project progressed steadily and was conceptually easier than the design phase. Since the project already had a concrete design, it was easy to begin the implementation without much thought.

The implementation phase involved a lot of trial and error, but working from a detailed plan was much easier. The components of the system are small, which makes it easy to implement one class and test its functionality in isolation.

As predicted in the design stage the programming was slow, but by spend-
8.2 IMPLEMENTATION PHASE

![Gantt Chart Diagram]

**Figure 8.1: Term 1 Gantt chart.**

...adequate time on the tasks the implementation was completed with plenty of time left to spare. Programming provided a challenge and the opportunity to learn and try new techniques. Most of low level work was use of data structures and simple network programming. At a higher level, the tree algorithms used by the collective operations provided the biggest challenge. Working from Pachecho’s tree broadcast implementation in [31] made this process easier, but the reduction algorithm was implemented with no reference material.

Table 8.2 contains the tasks for the latter half of the project, including all implementation tasks. Figure 8.2 presents these same tasks visually. Implementation began ahead of schedule, happening in parallel with the design phase of the project. By week 10 of the project, all initialisation routines had been completed, along with the basic network routines implemented in `NetDevice`. Also completed during this time were some of the MPI functions: `Send`, `Recv` and `Bcast`.

Following the Christmas vacation, implementation work was resumed. In weeks 14 to 18 the remainder of the project was completed. This remaining work only included the `Reduce` and `AllReduce` operations, and the corresponding `MpiOps`. Once implementation work on the project was completed, the remaining time was spent both writing and running the test programs detailed in Chapter 7.
The project management throughout the implementation phase was satisfactory. Although many of the tasks were slow to complete, by starting early and spending an adequate amount of time on each feature of the system, I was able to complete the implementation within schedule.

### 8.3 Performance Evaluation Phase

The performance evaluation of the project involved developing, and then running, the three test programs: ping-pong, Floyd-Warshall, and SWEEP3D. The overall goal is simple, yet the work involved was underestimated. The main problems with the performance evaluation were the lack of experience with C and large scientific codes, and the lack of a clear plan for the development and testing of these programs.

The three test programs already existed as C programs, meaning they only needed to be compiled, run, and the results recorded. Since latteMPI is a new implementation of MPI for Java, the programs needed to be converted from C to Java by hand.

The first two test programs, ping-pong and Floyd-Warshall, were developed quickly from the existing C implementations. Ping-pong consists of around 50 lines, and only makes use of the simple `Send` and `Recv` methods, so it was easy to create an equivalent Java program from the C version already available. Floyd-Warshall was slightly more complex, as it involved
some memory allocation for arrays. Memory management, particularly of arrays, differs in C and Java. Most of the work converting the C code to Java involved rewriting the code that was allocating memory. Since array addressing and the simple arithmetic statements needed for Floyd-Warshall are practically identical in C and Java, once the allocation code was modified, this was all the work that was needed to convert Floyd-Warshall.

SWEEP3D was the most difficult program to develop, even though, like the other programs, it was translated from an existing C implementation. The main difficulty in the conversion of SWEEP3D was the sheer size of the code. SWEEP3D consists of over 2000 lines of complex C code, filled with many idiomatic C features that do not map one-to-one into Java code. As with Floyd-Warshall, the first thing changed was the memory allocation. Following that, the MPI calls were translated into their equivalent latteMPI calls. Since the MPI Standard is so well defined, this was a simple task as it only involved slight alterations to the parameters. For example, the C calls take a \texttt{count} parameter, specifying how many elements to send, whereas latteMPI just sends the whole array. Sections of arrays can be created on-the-fly in Java\footnote{Using the \texttt{Arrays.copyOfRange()} method.}, so whilst this doesn’t affect the semantics, the syntax of the calls does need to be modified.

Whilst translating the MPI calls, the most complex issue was discovered: SWEEP3D relies on being able to send 3-dimensional arrays. In the C programming language these arrays are identical to one-dimensional arrays, they consist of a number of contiguous elements in memory. In Java, 3-dimensional arrays are represented as arrays of arrays. This means a 2-dimensional array is a 1-dimensional array where each element is itself a 1-D array. Since this is so semantically different from C arrays, a wrapper class was created in Java. This class provided an object that used a 1-dimensional array, analogous to the memory in C, to store the data, but provided methods to get and set values by providing 3-dimensional, \((i,j,k)\), co-ordinates. With this new wrapper class, the conversion of the MPI calls could be completed, with statements utilising these new objects modified to use the new accessor methods.

After the MPI code had been converted, only a few small tasks remained, mainly relating to the input and output, ensuring that the Java version could read the same input file format that the C program used. Finally, the arithmetic code was checked, but was generally copied straight from the C code to the Java version.
The conversion of the test programs proceeded erratically, and took longer than expected. Whilst ping-pong and Floyd-Were completed by Week 17 of the project, SWEEP3D was not completed until Week 19, despite the conversion process having begun in Week 12. The progress of the conversion was no doubt hindered by lack of any clear goals, and made more difficult when the problems described above were encountered.

Once all the test programs had been written, the second phase of performance evaluation began, running the tests. The tests are described in Chapter 7, so here we just present the issues related to the running of these tests.

In order to generate the data required for a fair comparison, the programs were required to be run on a number of different configurations – combinations of node counts and input sizes. Each job needed to be submitted to the cluster queue, in order to be run by the job scheduler. Each submission needed a separate script specifying the commands to run and upon submission, a number of parameters needed to be entered, such as the number of nodes and the runtime required. Creating the scripts and submitting each of the jobs manually would be both tedious and inefficient, so a testing script was developed.

The testing script for each job automatically runs the appropriate program on all desired configurations. The timing output is collected and stored in a log file with a name corresponding to the program and configuration used. An example test script is included in Figure 8.3.

In hindsight it is easy to see that the performance evaluation phase needed a more concrete plan, with more specific goals and deadlines to ensure that the work was completed in a timely fashion. Nevertheless, the work was complete within the time available. It is possible, however, that the work involved in the performance evaluation phase extended the scope of the project too far. The work carried out under the guise of performance evaluation could be extended to form a separate project, with a little extra work. The difficulty of translating a large C program to Java was underestimated. It required a lot more hand-coded alteration than initially expected, this made the task a lot more complex and a lot slower.

8.4 Development Process

The development process is an important feature of any software project, and latteMPI was no different. The specification (in Appendix A) described an agile style of development. Rather than focusing on traditional software devel-
8.4 DEVELOPMENT PROCESS

```perl
#!/usr/bin/perl -w

$shortLaunchString = "java -jar latteMPI-1.0-SNAPSHOT.jar";
$initCommands = "cat \$PBS_NODEFILE > hosts; sort -u hosts > hosts.txt";
$runInit = '"$initCommands';

for ($n = 1, $p = 1; $n < 9; $n = $n + 1, $p = $n * $n)
{
    $output = "/.sweep75-"."$n."x"."$n;
    $outputFile = $output.".out";

    print \nLAUNCHING: [".$launchString"]\n'';
    $programLog = '"$launchString';
    print \nCOMPLETED: [".$launchString"]\nDumping results ...\n'';
    $logFile = ">>"."$output.".log'';
    open MYFILE, $logFile;
    print MYFILE $programLog;
    close (MYFILE);
}
```

Figure 8.3: Job submission script used for the SWEEP3D program.

Development models such as the waterfall, agile methodologies are more developer
than business oriented.

The focus of agile development is rapid development and responding
quickly to changes in the specification. The Agile Manifesto [41] also lists cus-
tomer collaboration as a key point. Since the latteMPI project consists solely of
one member playing both the role of customer and developer, this is an easy
point to adhere to.

The advantages of an agile method come from being able to react quickly
to changes in the specification of the project. In a smaller project such as this,
these benefits were not as obvious as they would be in a large project with
many team members. This afforded extra flexibility as there were no customer
or company imposed constraints on the development process.

latteMPI followed some of the traditional development stages, starting
with the design phase. However, once the implementation phase of the project
started the agile techniques were applied more faithfully.

The pattern followed by the implementation was: implementing a func-
tion, unit testing the function, and then testing the function with a real program. After each mini-phase of the implementation a new function would be chosen and the process would start again. The order that the functions were implemented in is made clear in Chapter 5.

8.5 Tools

Some tools are a very user-specific choice, whilst others are imposed on the project by its definition. This section presents the tools used during the implementation and testing phase of the latteMPI project. The important features of the tools used will be highlighted, providing a sense of their contribution to the successful completion of the project.

8.5.1 Java

A critical component of any Java MPI implementation is the Java Development Kit (JDK). The JDK provides resources for developing and running Java programs. There is a distinction here between the runtime environment (described below), and the development kit. The JDK provides tools such as javac, the Java compiler, and jdb, the Java debugger. latteMPI was developed using the 1.6 version of the Sun JDK.

Due to the nature of Java, the project was run on a number of Java Virtual Machines (JVMs). During development, both Sun’s Java Runtime Environment (JRE) and the JDK VM were used. On the cluster used during testing, the OpenJDK Server JRE was used. The various JVMs used during the development and testing of latteMPI show that it is truly cross-platform, compiling and running without error on all manner of hardware and software combinations.

8.5.2 Build System

Maven provides the build system used by the latteMPI project. Maven, developed by Apache, is a Java oriented build system that favours convention over configuration [2]. This means that all of the source code and resources needed for latteMPI are organised in a particular hierarchy, one that is common to all Maven projects.
The advantage of using a build system, and Maven in particular, was the ease with which a jar file could be generated, containing all the classes needed in order to run user programs with latteMPI. Maven also provides integration with the JUnit unit test framework, which was covered in more detail in Section 6.1.

Another useful feature Maven provides is the ability to generate a site for the project. This generates a complete website, containing basic project information as well as Javadoc (documentation generated from comments) and unit test results.

8.5.3 Version Control

Version control is useful in a large software project to control changes made to the project, and source code in particular. Nothing is more frustrating than breaking the project and finding it very difficult to revert to a working version.

latteMPI used the git [11] version control system (VCS). The most useful feature is branching, which is the ability to have a parallel version of the project files with different changes made to them. If the changes are successful then the branch can be merged back into the main source code and development can continue.

The tree algorithms discussed in Chapter 5 made lots of use of the branch feature. Whilst the algorithms are conceptually quite simple, a number of implementation pitfalls meant that they were developed in a separate branch and only added to the main code once they had been tested.

8.6 Conclusion

The tools used throughout the project have been carefully selected and evaluated to ensure their suitability. Using a build system helped with the initial development of the project, and version control has proved useful for prototyping ideas during the implementation phase. Unit testing has been a crucial feature of the development of the project, enabling a number of simple bugs to be caught before the project was even used, since all the tests were run automatically by Maven when the project was compiled.
### Term 1

<table>
<thead>
<tr>
<th>Week</th>
<th>Tasks</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Start writing project specification. Begin design</td>
</tr>
<tr>
<td>2</td>
<td>Start to sketch a map of the classes to be used. Submit specification</td>
</tr>
<tr>
<td>3</td>
<td>Continue to resolve design issues</td>
</tr>
<tr>
<td>4</td>
<td>Begin implementation</td>
</tr>
<tr>
<td>5</td>
<td>Begin progress report</td>
</tr>
<tr>
<td>6</td>
<td>Get initial code structure in place. Work out issues relating to version control and build system. Submit progress report</td>
</tr>
<tr>
<td>7</td>
<td>Continue with implementation. Start unit testing.</td>
</tr>
<tr>
<td>8</td>
<td></td>
</tr>
</tbody>
</table>

### Term 2

<table>
<thead>
<tr>
<th>Week</th>
<th>Tasks</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>Begin writing report (first draft)</td>
</tr>
<tr>
<td>16</td>
<td>Complete implementation. Begin testing</td>
</tr>
<tr>
<td>17</td>
<td>Start performance analysis</td>
</tr>
<tr>
<td>18</td>
<td>Start preparing presentation</td>
</tr>
<tr>
<td>19</td>
<td></td>
</tr>
<tr>
<td>20</td>
<td></td>
</tr>
<tr>
<td>21</td>
<td></td>
</tr>
<tr>
<td>22</td>
<td></td>
</tr>
<tr>
<td>23</td>
<td></td>
</tr>
<tr>
<td>24</td>
<td></td>
</tr>
</tbody>
</table>

### Term 3

<table>
<thead>
<tr>
<th>Week</th>
<th>Tasks</th>
</tr>
</thead>
<tbody>
<tr>
<td>25</td>
<td>Complete any unfinished implementation work.</td>
</tr>
</tbody>
</table>

Table 8.2: Project Timetable: Implementation Phase
This brief chapter will summarise my thoughts on the project using responses to the questions that make up the paragraph titles. A more comprehensive critique of the work undertaken is presented in Chapters 8 and 10.

**What is the technical contribution of this project?** The project presents an MPI implementation written using pure Java code. An inherent feature of Java is the fact that byte-code compiled on one machine can be run by the Java Virtual Machine of any other computer, even if they have no similarities other than an available Java implementation.

The project is both simple and designed in a way that will be familiar to many students who have learned object-oriented programming techniques. It therefore provides an ideal learning environment when discussing the development of systems, message-passing or otherwise.

**Why should this contribution be considered either relevant or important to computer science?** The portability of the Java language (described above) is new in parallel programming, as programs typically had to be ported manually to other systems. By removing the need for manual porting, developers can focus simply on providing a JVM for their hardware, confident in the fact that there are existing parallel programs can be run instantly on the new hardware, without requiring further tweaking.
How can others make use of the work in this project? Others can make use of the work presented here in two ways. Firstly, by using the parts of the MPI Standard that latteMPI implements to create and run numerous parallel programs on a wide variety of hardware. Secondly, extension of the system with more operations – non-blocking communication for example, would make an excellent project for someone interested in message-passing or high-performance computing.

Why should this project be considered an achievement? Creating thousands of lines of code is an achievement on its own, but by creating a usable, useful project I feel I have surpassed my expectations. Running my first test programs (successfully) on a cluster of machines is definitely an experience to remember.

The project fulfills its objectives and the end result is a simple and effective tool that can solve real problems. The test programs written demonstrate the success of the project, and show that a tool such as an MPI implementation, once constrained, can be developed by an undergraduate computer scientist.

What are the weaknesses of this project? The limited subset of the MPI Standard that has been implemented could be seen as a weakness, but it has provided enough scope for writing some interesting test programs. An increase in the flexibility of the configuration of the project would be a nice improvement, since some variables that could be user configured are currently hardcoded in a manner tuned to the system in the Department of Computer Science.
This chapter evaluates the project with respect to both the functional and non-functional requirements detailed in Chapter 3. Accompanying each requirement is a discussion on whether the requirement was fulfilled and how.

10.1 Evaluation With Respect to the Functional Requirements

The functional requirements of the system are those that can be objectively measured, and are vital components of a successful project. Each requirement makes up a heading, and below each heading is a discussion on whether the requirement was successfully achieved.

10.1.1 Provide a usable subset of the MPI Standard

1.(a) Management functionality, to start and end programs correctly.

Completed The MpiStarter class reads any needed configuration information using command line parameters, before using ssh to start an MpiProcess on each required machine. The MpiProcess then initialises the MPI classes before loading the user program with reflection. Once the user program completes, its exit status is examined and returned via ssh to the MpiStarter class that is waiting on the root node. In the event of premature process termination, all child processes are shutdown, to ensure no processes
are left running on remote nodes.

1.(b) Point-to-point communication routines.

**Completed** Two point-to-point communication routines have been provided: `Send` and `Recv`. These routines function as their names suggest. They provide the simplest form of message passing by building on top of the networking functions in the `NetDevice` class. Despite being simple, these two routines can be used as primitives for building the more complex collective routines.

1.(c) Collective communication routines.

**Completed** Three collective communication routines are provided. They are `Bcast`, `Reduce` and `AllReduce`. By building on top of the point-to-point communication methods, these collective routines were easy to implement. However, they use more complex algorithms in order to improve performance over naïve implementations of the same function.

10.1.2 Implement the subset according to the MPI Standard

2.(a) Functions semantically equivalent to the MPI Standard.

**Completed** The functions are semantically equivalent to those found in version 2.2 of the MPI Standard. This equivalence has been verified both through running programs with latteMPI and OpenMPI and verifying the results, as well as using automated unit tests to ensure corner cases behave correctly.

2.(b) Processor count independent.

**Completed** The number of processors to use is specified in a parameter on the command line and passed to `MpiStarter`, which then starts that number of processes. The only limit on the number of processes that can be started is the maximum number of Java threads and the maximum number of `ssh` connections. Both these limits are far higher than would be reached during normal use. The implementation has been successfully tested on up to 64 processors.
10.1.3 Provide an implementation written in Java

**Completed** All of the latteMPI source code is written in Java, and the only external program it relies on is ssh. All MPI functions can be accessed from the user program using Java’s include syntax. The fact that latteMPI relies on the ssh program will not cause problems during usage, since ssh is a standard program that comes pre-installed on all Linux systems – the operating system of choice for most small computer clusters – the type that latteMPI is most suited to.

10.2 Evaluation With Respect to the Non-Functional Requirements

The non-functional requirements are the parts of the specification that cannot be measured objectively. For each requirement, a decision on whether it has been adequately completed is followed by evidence to support the decision.

10.2.1 Acceptable performance

**Completed** The latteMPI project provides an acceptable level of performance, given the constraints placed upon the project by Java, development time, and developer experience. Chapter 7 shows that whilst not being as fast as C and the Open MPI implementation, latteMPI can still be expected to produce results in a similar time frame, even for complex computational programs like SWEEP3D. The advantages of Java help to counter some of these performance criticisms, but there is no escaping the fact that Java adds a lot of overhead, especially to low-level tasks such as marshalling data for packing into messages. These low level tasks are critical in an MPI implementation, and the overhead of Java accounts for approximately 70 percent of the time overhead in the ping-pong benchmark.

As an undergraduate project, the performance of latteMPI is acceptable, but there are a number of factors that could be explored for improving the performance if latteMPI was to be used in a more mission-critical setting. Some compilers exist which convert Java code to machine code, eliminating the overhead of the virtual machine. However, latteMPI would then be made as portable as the existing C implementations of MPI. The JVM has a number
of performance parameters that can be adjusted, which may provide performance increases without having to resort to compiling to machine code.

### 10.2.2 Extensible component architecture

**Completed** The NetDevice interfaces defines the simple network methods that latteMPI relies on for data communication. This interface can be replaced with any class that implements these functions, allowing latteMPI to take advantage of hardware specific classes.

### 10.2.3 Testable

**Completed** All code in the project is tested automatically with JUnit unit tests. These tests exercise class interfaces as well as internal methods ensuring all code performs as expected. The unit tests cover 71 percent of all lines of code in the project, ensuring that not only is the code testable, but that it is tested.

### 10.2.4 Documentation

**Completed** All methods, both public methods visible to the user and internal methods used only by other latteMPI classes, are commented using the Javadoc style. Documentation webpages are generated from these comments using the javadoc tool. Maven’s site function is used to build these documentation pages and other information into a complete webpage containing all the project documentation\(^1\).

### 10.2.5 Coding Standards

**Completed** Through the use of editor settings, all code uses a consistent style. All files use 2-space indentation and Kernighan & Ritchie style braces, where the opening brace begins on the same line as the statement preceding it. As well as detailed Javadoc comments, instructive comments are placed throughout the code, particularly at points where the action performed by the code is not obvious. All files are managed using the git version control software, enabling easy rollback of any erroneous changes. All these properties

---

\(^1\)This webpage is included on the submitted CD
make the project easy to work with, and leave it accessible to any future developers.
CHAPTER 11

CONCLUSION

This final chapter summarises the main results achieved by the latteMPI project in terms of its performance, as well as its success from a project management point of view.

11.1 Project Context

The latteMPI project nestles in to the landscape of Java MPI implementations next to MPJ Express, another implementation taking a pure Java approach to implementing the MPI Standard. Whilst latteMPI does not implement all functions, those that it does implement work correctly and efficiently. Also similar are the C and Fortran implementations we have seen, in particular Open MPI, which strongly favours a modular design, much like latteMPI does.

The work that extends these C and Fortran implementations is just as applicable to latteMPI; the language independent MPI Standard ensures this. It would be interesting to try and implement extra features for latteMPI, to see if Java has any strengths that would make this easier than the corresponding extension of a C or Fortran implementation.

11.2 Performance Summary

Java is not the fastest computer programming language, and latteMPI is not the fastest MPI implementation. However, the performance across the benchmark and test applications is both satisfactory and acceptable. Also, it is worth
noting the extra administration-type overheads that Java eliminates: compilation time, setup time, dependence on machine architecture. Eliminating these reclaims some of the time that Java loses with its slower performance.

11.3 Project Experiences

The project has been a particularly enjoyable way to spend the past 6 months. The goal of the latteMPI is well defined, meaning it was easy to start designing the project from the outset.

The tools used on the project have contributed to it’s design and success. Using JUnit tests was a revelation, and something that should be applied to every project. Harnessing the power of a build system like Maven allowed easy deployment from the development machine to the test cluster and also produced a feature-full project website containing information and documentation for latteMPI.

11.4 Closing Remarks

The latteMPI project acheived its goal of successfully implementing a subset of the MPI Standard. The subset implemented has proven to be usable as well as adequate, providing enough scope to write a number of non-trivial test programs. Whilst performance could be better, Java has some redeeming qualities that make it a fun language to work with. The project has been challenging, yet enjoyable, and it is great to complete it with a solution to be proud of.


[33] Rahimian, A., Lashuk, I., Veerapaneni, S., Chandramowlishwaran, A., Malhotra, D., Moon, L., Sampaith, R., Shringarpure,


Appendices
APPENDIX A

SPECIFICATION

A.1 Problem

The Message Passing Interface (MPI) is now an ubiquitous standard in writing HPC applications. Existing implementations focus on delivering outstanding performance in C and Fortran [18], and the MPI Standard [27] defines bindings for both C and Fortran-90. There is no single binding defined for Java, although a number of projects currently provide Java implementations of the specification\(^1\). Java is a cross-platform language whose performance is continually increasing with JVM updates; a Java implementation of the MPI Standard would be a powerful tool in high performance computing, providing truly portable high performance applications.

A.2 Objectives

The main objective of the project is to implement a subset of the MPI functionality in Java. This will involve providing the MPI interface [27] to Java programs and implementing the routines necessary to perform point-to-point communications and collective operations. The project will also be able to initialise and finalise jobs using the semantics defined in the specification [27]. This will enable integration with existing job schedulers.

\(^1\)These implementations are mpiJava [10] and MPJ-Express [6]. mpiJava uses JNI to provide Java wrappers to the C implementations of MPI, MPJ-Express provides both a pure Java implementation as well as the option to use a native implementation.
There are a number of clear objectives which each represent a significant step towards the final goal:

- Design class structure and interface to be provided.
- Implement initialisation routines to set up the runtime environment.
- Implement communication routines used by MPI operations.
- Measure performance in comparison to existing C and Fortran implementations.
- Write a report detailing the design and implementation of the project.

The project will create a pure Java implementation of a subset of the most common MPI operations.

### A.3 Methods

#### A.3.1 Design

The design of the implementation will need to be carefully considered so that it can provide a concrete interface with which users can program. The ultimate work product of this objective will be a description and class diagram detailing all of the components that will provide the interface.

#### A.3.2 Implementation

**Programming** The implementation is largely an exercise in software development, so the choice of methodology is important. The code will be developed independently, so will not use a rigid process but instead focus on the well-documented agile techniques as they apply to a single-person team. The project will aim to follow the timetable (see Subsection A.4) and implement the software from the top down. By first creating all of the classes, and methods using default return values, I can then work down, adding the code to each element. In this way, the interface will compile and run as quickly as possible, but will leave scope for further optimisation if there is time.
Testing  Unit testing will be used to ensure that the components all fulfill their documented requirements. Testing will use the JUnit [21] framework, which enables simple tests to be written in Java. Parallel testing will need to be performed on the clusters located in the Computer Science Department (see Subsection A.5).

Build System  To manage the project, Apache’s Maven [2] software will be used. Maven’s build system provides a one step way of compiling the project and comes with a plug-in to enable automated unit testing using the JUnit framework. Maven can also generate a project website, including the documentation and test results.

A.3.3 Performance Analysis

Performance analysis will involve monitoring the time certain tasks take in comparison to other implementations of MPI, both in Fortran and C. The wall-clock time will be measured for performing simple tasks such as small message ping-pongs, as well as executing a larger, functional MPI program. The results will then be plotted and included in the report. To provide a real-world benchmark, an existing C code will be converted to Java. This Java code will then be compared to the C version.

Performance testing of the implementation against other Java MPI implementations – mpiJava and MPJ-Express – will also be undertaken; this will allow for a better comparison of the performance. The overhead for these implementations will be largely the same, while the overhead for both C and Fortran will be low, giving it a significant performance advantage.

A.3.4 Project Report

The project report will be the vehicle used to convey the work that has been carried out during the design and implementation phase of the project.

Writing the first draft in parallel with the implementation of the project will allow the report to accurately convey what has happened during the course of the project.
A.4 Timetable

Above (Figure A.1) is a timeline outlining the work to be undertaken during the project; it provides a rough outline of the estimated time required for each component. Note the overlap between the three code-related components: design, implementation, and testing. This is related to the agile methodology, but also to the fact that there is a strict time constraint and the work must be done both quickly and accurately.

Table A.1 on page 108 is a table detailing the work to be done. The weeks are numbered using the University’s numbering scheme. Weeks left blank indicate a continuation of the previous weeks work.

The 4 weeks of the Christmas vacation are to be left un-timetabled, leaving time for revision and other coursework. Spare time at Christmas can be used as a buffer in the timetable, to ensure that the second term work starts on schedule.

Easter will also be used for revision and other coursework, but time may be allocated for finalisation of the project report.

A.5 Resources

During the testing phase of the project the PCAV/HPSG clusters in the Computer Science department will be used to run the software, testing whether the
inter-node communication portion of the implementation will run correctly. The job will need to be scheduled on the cluster, so it will be important to ensure that this is started in plenty of time in case there is waiting time or unexpected down-time. The implementation will need to be compatible with PBS/MAUI schedulers so that it can be run on the cluster without interfering with other jobs and production work.

A.6 Legal, Social, Ethical, and Professional Issues

A.6.1 Social and Professional

The MPI Standard has been built up over a number of years and is widely regarded as the industry standard interface for message passing, so it is important to provide an accurate implementation of the described bindings. Users will be expecting the implementation to function in a standard way, consistent both with the bindings and also with other implementations, Java or otherwise.
### Table A.1: Project Timetable

<table>
<thead>
<tr>
<th>Week</th>
<th>Tasks</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Start writing project specification. Begin design</td>
</tr>
<tr>
<td>2</td>
<td>Start to sketch a map of the classes to be used. Submit specification</td>
</tr>
<tr>
<td>3</td>
<td>Continue to resolve design issues</td>
</tr>
<tr>
<td>4</td>
<td>Begin implementation</td>
</tr>
<tr>
<td>5</td>
<td>Begin progress report</td>
</tr>
<tr>
<td>6</td>
<td>Get initial code structure in place. Work out issues relating to version control and build system. Submit progress report</td>
</tr>
<tr>
<td>7</td>
<td>Continue with implementation. Start unit testing.</td>
</tr>
</tbody>
</table>

(a) Term 1

<table>
<thead>
<tr>
<th>Week</th>
<th>Tasks</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>Begin writing report (first draft)</td>
</tr>
<tr>
<td>16</td>
<td>Complete implementation. Begin testing</td>
</tr>
<tr>
<td>17</td>
<td>Start performance analysis</td>
</tr>
<tr>
<td>18</td>
<td>Start preparing presentation</td>
</tr>
<tr>
<td>19</td>
<td></td>
</tr>
<tr>
<td>20</td>
<td></td>
</tr>
<tr>
<td>21</td>
<td></td>
</tr>
<tr>
<td>22</td>
<td></td>
</tr>
<tr>
<td>23</td>
<td></td>
</tr>
<tr>
<td>24</td>
<td></td>
</tr>
</tbody>
</table>

(b) Term 2

<table>
<thead>
<tr>
<th>Week</th>
<th>Tasks</th>
</tr>
</thead>
<tbody>
<tr>
<td>25</td>
<td>Complete any unfinished implementation work.</td>
</tr>
<tr>
<td>26</td>
<td>Finish final draft of report.</td>
</tr>
<tr>
<td>27</td>
<td>Send report for proofreading.</td>
</tr>
<tr>
<td>28</td>
<td>Make corrections to report.</td>
</tr>
<tr>
<td>29</td>
<td>Second proofreading.</td>
</tr>
<tr>
<td>30</td>
<td>Miscellaneous corrections.</td>
</tr>
<tr>
<td>31</td>
<td>Submit project report.</td>
</tr>
</tbody>
</table>

(c) Easter and Term 3
This appendix includes comprehensive instructions on compiling and using latteMPI, as well as a brief section on writing a simple message-passing program.

### B.1 Setup

The setup of latteMPI has two parts. The first is ensuring that your environment is setup ready to run programs. The second is compiling the latteMPI library to include the correct constants for running on your system.

#### B.1.1 Environment

latteMPI relies on ssh for invoking processes, so it is important to ensure it is configured correctly prior to using the program. The first step is to make sure that the sshd program is running on the nodes you intend to use for running latteMPI. Setting up sshd varies between operating systems, so it is best to follow an appropriate tutorial on the internet, or ask a local expert.

Once sshd is running, you will need to set up keys so that latteMPI can ssh to remote nodes without needing a password entered. Again, this process varies from system to system, but generally your key will need to be generated then copied into .ssh/authorized_keys, in your home directory. To check that everything is working as it should be, try and connect to one of the nodes using ssh; if you can connect without a password then latteMPI should be
able to as well.

### B.1.2 Compilation

Since latteMPI uses the Maven build system, compilation is fairly easy. Download a version of Maven for your operating system from [2] and install it. Use the command

```shell
maven clean install
```

to generate the `latteMPI.jar` file in the `target` folder. Copy this to the directory you will be running your programs from, and you are ready to proceed to the next section.

### B.2 Programming with latteMPI

In this section we will see how to write a simple MPI program using latteMPI. The program will send a message from one root process to every other process, each process will then print out it’s message. This program is the message-passing equivalent of “Hello, World!”.

The `import` statement in line 1 is needed to include all the necessary class files for compiling our code.

Two useful functions of any MPI implementation are `Size` and `Rank`. These functions return the number of processes and the rank of this particular process. The main use of the rank is for switching to different blocks of code depending on which processor the program is executing on.

You can see this on line 19 where the program branches. The process with rank 0, normally called the `root` process, will being executing the statements inside this block. First it will print out a message, using the `println` command, this works exactly the same as in a normal Java program. After this we enter a for loop. By starting at 1 and executing n-1 iterations, we can send a message to each of the other n-1 processors, using the `i` variable to identify the rank of the process to send to next. The send happens in line 24. As with all MPI functions, the method is in the class `Mpi`, since we are using the default communicator, we access this through `COMM_WORLD`. We then call the `Send` method with its required parameters: data to send, rank, and message tag. Here we can see a useful feature of Java, creating the integer array to send without having to allocate it to a variable first. Whilst latteMPI can only send
```java
import uk.ac.warwick.java mpi.*;

public class HelloWorldMPI {
    public static void main(String[] args) {
        Mpi.Init(args);

        // Find out rank of this processor
        int rank = Mpi.COMM_WORLD.Rank();
        // Find the size of the communicator
        // (the number of processes we are using)
        int size = Mpi.COMM_WORLD.Size();

        /*
         * If we are process 0 (the "root" process), then we
         * branch into this if statement and will send a message
         * to each of the other processes.
         */
        if (rank == 0) {
            System.out.println("Hello from process 0, starting sends....");
            for (int i = 1; i < size; i++) {
                // Send a message to each process, containing a single
                // integer that is that processes rank.
                Mpi.COMM_WORLD.Send(new int[]{i}, i, 0);
            }
        } else {
            // Create array to receive the message
            int[] message = new int[1];
            // Create status object, to check the status of theRecv call
            Status status = new Status();

            // Receive the message with tag 0 from process 0, and store it
            // in the array message
            Mpi.COMM_WORLD.Recv(message, 0, 0, status);

            /*
             * This print statement is a sanity check,
             * the expected output is of the form:
             * "Hello from process 3, I received 3"
             */
            System.out.printf("Hello from process %d, I received %d \n", rank, message[0]);
        }
        Mpi.Finalize();
    }
}
```

Figure B.1: Simple “Hello World” program using MPI.
arrays of primitives, this makes it easy to put just a single value into an array that can then be sent.

All other processors will have a rank greater than 0, so will execute the else part of this block. Here the processors have to create an array in which to receive the message. This is necessary because the 

**recv** method will copy the message into that array. A **Status** object is created on line 30. We met this object in Chapter 5, so we know it is used to return information about the received message. The call to **recv** is where the data gets transferred into the user program. Like **send**, this function is in the **COMM_WORLD** object of the **Mpi** class. The parameters are: the array to copy the data into, the source of the message, the tag of the message, and a **Status** object. When this method is called, the data is copied to the **message** array, and information is added to the status object. We know that the message will be sent from process 0 and will have tag 0 (since this was the tag specified in the **send** on line 24).

Once the data has been received, each process will perform a **printf** statement, we can use this to check everything worked as expected. Each process will print out its rank, as well as the message it received from process 0. The message should be the same as the rank.

### B.2.1 Compilation

If we save this source code in the file **HelloWorldMPI.java**, then we will compile it with the command line:

```
javac -cp latteMPI.jar HelloWorldMPI.java
```

After compilation the file **HelloWorldMPI.class** will exist, this is what we will pass to latteMPI in order to run the program.

### B.2.2 Running programs

Running programs with latteMPI works in much the same way as any Java jar file. Assuming latteMPI has been compiled and set up as described in Section B.1 then running a program on any number of processors is very easy:

```
java -jar latteMPI.jar -hostfile <path to hostfile> \n   -n <number of processors> -className <name of program>
```

The important arguments are **-hostfile**, which needs to be an absolute path to the file containing the host names of the computers you wish to use. The
The next argument `-n` is simply an integer specifying how many processors you wish to use. Finally the `-className` argument needs to be the name of the program you wish to run. By default latteMPI will look for your ssh key file in `~/.ssh/id_dsa`, if your key is somewhere else, then pass the additional argument `-ssh <path to key>` to latteMPI when you run it. If we put all this together, an example command to run the program from Section B.2 would be:

```
java -jar latteMPI.jar -hostfile /home/latteMPI/hosts.txt -n 4 \\
-className HelloWorldMPI
```

Note that the `\` means this command should be all one line. However, it needed to be spread over multiple lines for display in this document.

## B.3 Further Information

For further details on latteMPI and the functions it provides please see the javadoc documentation included on the accompanying CD.

For more information on programming with MPI please see [31, 16], two books that contain excellent introductions to parallel programming and MPI in particular.
APPENDIX C

PRESENTATION SLIDES
latteMPI

A pure Java subset of the MPI Standard

Background
Design
Implementation
Performance
A pure Java subset of the MPI Standard
A pure Java subset of the MPI Standard

Background
Design
Implementation
Performance

A pure Java subset of the MPI Standard
A usable subset ...

Point-to-point

Collective

Point-to-point

•Send
•Receive
Collective

- Broadcast
- Reduce
- Allreduce
Background
Design
Implementation
Performance

Ping-pong
Results: Ping-pong

![Graph showing results for Ping-pong]

Results: Floyd Warshall

![Graph showing results for Floyd Warshall]

C.0
SWEEP3D

Results: SWEEP3D

A pure Java subset of the MPI Standard