Parallel Discrete Event Simulation on the Grid

James Thomas
Computing (Industry) BSc
2005/2006

The candidate confirms that the work submitted is their own and the appropriate credit has been given where reference has been made to the work of others.

I understand that failure to attribute material which is obtained from another source may be considered as plagiarism.

(Signature of student)__________________
# Acronyms

<table>
<thead>
<tr>
<th>Acronym</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>API</td>
<td>Application Programming Interface</td>
</tr>
<tr>
<td>ATM</td>
<td>Asynchronous Transmission Mode</td>
</tr>
<tr>
<td>CPU</td>
<td>Central Processing Unit</td>
</tr>
<tr>
<td>DAME</td>
<td>Distributed Aircraft Maintenance Environment</td>
</tr>
<tr>
<td>FEL</td>
<td>Future Event List</td>
</tr>
<tr>
<td>HTTP</td>
<td>Hyper Text Transfer Protocol</td>
</tr>
<tr>
<td>IP</td>
<td>Internet Protocol</td>
</tr>
<tr>
<td>LAN</td>
<td>Local Area Network</td>
</tr>
<tr>
<td>LP</td>
<td>Logical Process</td>
</tr>
<tr>
<td>MPI</td>
<td>Message Passing Interface</td>
</tr>
<tr>
<td>OGSA</td>
<td>Open Grid Services Architecture</td>
</tr>
<tr>
<td>PC</td>
<td>Personal Computer</td>
</tr>
<tr>
<td>PDES</td>
<td>Parallel Discrete Event Simulation</td>
</tr>
<tr>
<td>RAM</td>
<td>Random Access Memory</td>
</tr>
<tr>
<td>RSL</td>
<td>Resource Specification Language</td>
</tr>
<tr>
<td>SSH</td>
<td>Secure Shell Server</td>
</tr>
<tr>
<td>SOAP</td>
<td>Simple Object Access Protocol</td>
</tr>
<tr>
<td>TCP</td>
<td>Transmission Control Protocol</td>
</tr>
<tr>
<td>UML</td>
<td>Unified Modelling Language</td>
</tr>
<tr>
<td>UDP</td>
<td>User Datagram Protocol</td>
</tr>
<tr>
<td>VO</td>
<td>Virtual Organisation</td>
</tr>
<tr>
<td>WAN</td>
<td>Wide Area Network</td>
</tr>
<tr>
<td>WRG</td>
<td>White Rose Grid</td>
</tr>
<tr>
<td>WSDL</td>
<td>Web Services Description Language</td>
</tr>
</tbody>
</table>
Summary

Simulation is a very popular method for assessing the behaviour of real world systems, by modelling these real world systems and observing the outcomes of the models when ran under various conditions. When computers became popular in industry and the scientific community, the processing power they provided meant they were a very natural environment in which to run simulations. Computer simulation requires models that can be used to represent the real world systems in an appropriate format, so that they can easily make use of the computational resources available.

Parallel discrete event simulation is a modelling methodology that exploits the inherent parallelism in many real world systems, by splitting up the model into distinct entities. Each entity, or logical process, can then be run on separate computers in a cluster, sharing messages with standard parallel programming techniques, enabling the simulation to proceed at a much faster rate.

Grid computing is a new and emerging technology that aims to provide people with common interests, as known as virtual organisations, with access to enormous amounts of computational resources, connected using high speed data networks. Grid computing is therefore a very promising technology with regards to the world of computer simulations, possibly providing access to large scale processing power. This project will attempt to establish whether it is viable for parallel discrete event simulations to take advantage of this new resource pool to further increase the performance of simulations.
Acknowledgements

Firstly I’d like to thank Karim for his unwavering support and constant help, Pete Jimack for all the useful feedback and his guidance as my tutor at Leeds University over four years, John Hodrien for his technical support, a number of proof readers I badgered into reviewing my drafts till the early hours and Dad and Pam for their continuing emotional and financial support. Lastly, I’d like to thank Sir Alex Ferguson, a truly inspirational person.
# Contents

1 Introduction .................................................. 1
   1.1 Context .................................................. 1
   1.2 Aim ..................................................... 1
   1.3 Objectives .............................................. 1
   1.4 Minimum Requirements ................................. 2
   1.5 Deliverables ............................................ 2
   1.6 Project Management ..................................... 2

2 Background Research .......................................... 3
   2.1 Discrete Event Simulation ............................... 3
      2.1.1 Simulation and Modelling ......................... 3
      2.1.2 Modelling methodology ............................ 4
      2.1.3 Simulation models ................................ 5
         2.1.3.1 Petri Nets ..................................... 5
         2.1.3.2 Finite State Machines ........................ 5
         2.1.3.3 Queueing Model ............................... 6
      2.1.4 Parallel Simulation and causality errors ............ 7
      2.1.5 Dealing with causality error in parallel simulations ............................................. 8
      2.1.6 Conservative protocol .............................. 8
      2.1.7 Optimistic protocol ............................... 9
      2.1.8 Synchronous protocol ............................. 9
   2.2 Parallel Programming ................................... 10
      2.2.1 Introduction ....................................... 10
2.2.2 Parallel Architectures ................................................. 11
2.2.3 Message Passing Interface ........................................... 12
2.3 Grid Computing ........................................................... 13
  2.3.1 Beginnings of the Grid .............................................. 13
  2.3.2 Evolution of the Grid ............................................... 14
  2.3.3 Virtual organisations ............................................... 14
  2.3.4 Grid Architecture .................................................. 15
    2.3.4.1 Fabric Layer .................................................. 16
    2.3.4.2 Connectivity Layer .......................................... 16
    2.3.4.3 Resource Layer ............................................... 17
    2.3.4.4 Collective Layer ............................................. 17
    2.3.4.5 Application Layer ........................................... 17
  2.3.5 Grid middleware ................................................... 17
    2.3.5.1 Globus Toolkit ............................................... 18
  2.3.6 Grid Services Architecture ....................................... 19
  2.3.7 Parallel Programming and the Grid ............................... 19
  2.3.8 Examples Grids and their Applications ......................... 20
    2.3.8.1 White Rose Grid ............................................. 20
    2.3.8.2 DAME ....................................................... 21
  2.4 Summary ................................................................... 22

3 Design ........................................................................ 23
  3.1 Introduction .............................................................. 23
  3.2 Environment ............................................................. 24
    3.2.1 Software ............................................................ 24
    3.2.2 Hardware ............................................................ 25
  3.3 Measuring performance ................................................ 25
    3.3.1 Performance metrics .............................................. 25
      3.3.1.1 Time ............................................................. 26
      3.3.1.2 Data transfer .................................................. 26
      3.3.1.3 Load ............................................................ 27
5.2 Performance analysis .................................................. 49
  5.2.1 Main metric ....................................................... 49
  5.2.2 Secondary metrics ............................................. 53

6 Evaluation .............................................................. 55

  6.1 Performance of PDES on the Grid ............................... 55
    6.1.1 Performance evaluation against the main metric .......... 56
    6.1.2 Performance evaluation against secondary metrics ....... 56
    6.1.3 Evaluation of results against related research .......... 56
    6.1.4 Sources of error and problems encountered ............. 58
    6.1.5 Conclusions about the performance impact of PDES on the Grid .... 59
    6.1.6 Further enhancements ....................................... 61
      Wide Area Network .............................................. 61
      Change MPI Protocol .......................................... 61
      Number of LPs .................................................. 61
      Arrival and service rates .................................... 62

  6.2 Overall Project Evaluation ..................................... 62
    6.2.1 Project aim .................................................. 62
    6.2.2 Project objectives .......................................... 62
    6.2.3 Minimum requirements and extensions ..................... 63
    6.2.4 Project Management ........................................ 65
    6.2.5 Environment ................................................ 65
    6.2.6 Methodology ................................................. 66

Bibliography ............................................................ 67

Appendix A - Personal Reflection .................................. 70

Appendix B - Experimental Data and Graphs ...................... 74

Appendix C - Project Schedule Charts ............................. 81

Appendix D - UML Design Diagrams ................................. 82
Chapter 1

Introduction

1.1 Context

This project report details the evaluation of the performance of parallel simulations on a Grid test bed. The emerging Grid platform is a massive opportunity for access to large scale computational resources, parallel simulations have traditionally ran on clusters of high performance computers and being able to take advantage of these resources would be of great advantage in the simulation world. The project will construct a sample parallel simulation application, using standard parallel programming techniques, and compare the results of a series of experiments, using the application, when ran on a Grid test bed and a normal parallel programming environment.

1.2 Aim

The aim of this project is to evaluate the performance of parallel simulation on a GRID test bed.

1.3 Objectives

The main objectives of this project are to:
• Produce a software system implementing a parallel discrete event simulator using parallel programming in a Grid environment.

• Use this simulator to test performance of parallel discrete event simulation on the Grid.

• Evaluate parallel simulation on a Grid in comparison to traditional parallel programming environments.

1.4 Minimum Requirements

The minimum requirements of this project are:

• Implement a simple discrete event simulator to simulate a queueing network.

• Modifying the simple discrete event simulator to incorporate one parallel discrete event simulation protocol.

• Extend the parallel discrete event simulator to run on a Grid test bed.

The possible extensions are:

• Extending the simulator to run on a large scale wide area network Grid, such as the White Rose Grid.

1.5 Deliverables

The deliverables for this project are:

• The final project report.

• A software parallel discrete event simulator written to help with the performance evaluation.

1.6 Project Management

An initial project schedule was devised and is available in Appendix C, Figure 6.2.6. This chart models my initial time plan for the project and incorporates a series of milestones to help me gauge my progress.
At the start of this project a thorough and extensive period of background research was completed in order to gain an understanding of the problem domain in which I would be working. A summary of the areas I spent time carrying out background reading in is given in this chapter.

2.1 Discrete Event Simulation

2.1.1 Simulation and Modelling

Simulation “is the imitation of the operation of a real-world process or system over time”[1]. Simulations are performed using models of the real-world system. The models incorporate many of the components of that system in order to exhibit similar behaviour to that of the real world system or process. A simulation model would usually model particular aspects of a system that are of interest to us, so that its behaviour can be observed under certain conditions, mimicking real world scenarios, in order to ascertain what behaviour the real world system would exhibit. This is extremely useful in scenarios where applying the relevant conditions to a real world system isn’t possible, could be dangerous or the real world system hasn’t been built yet and we are testing for viability at the design stage.

Constructing a simulation model needs the user to be aware of the relationships between different elements of the system and the affect they have on each other. Simulation can be a very effective way of
researching a system’s probable behaviour, under certain unique conditions, but its accuracy depends on a number of factors and it isn’t always an appropriate methodology to follow. Conditions under which simulation might not be appropriate range from when solving the problem can be completed using common sense or analytically, when using direct models may be far easier, when the cost of simulation far exceeds the savings, there is no data available to model a system or a system is too complex to model effectively[1].

2.1.2 Modelling methodology

For a simulation to be effective the problem needs to be well defined and the relevant information available to construct a correct model. It can also be used on existing systems or completely new ones as a design guide or prediction tool. Modelling simulations have been used extensively in a broad range of industries such as construction and manufacturing with great success[1]. Modelling a system requires an understanding of the elements in a system, called entities, and their relationships with each other. One of the most important features of a model is deciding on the boundary between the system and its environment. This will decide the amount of scope to be tackled within the modelling simulation.

Entities within a system have attributes which are representative of the properties of each entity. An event within a system is a process within the system that changes the internal state of the system, represented by state variables[2]. All systems can be described as either continuous or discrete by their nature. Continuous systems have changes in their state variables happening continuously through their lifetime, whereas in a discrete system state variables only change at discrete time intervals[1]. Each discrete event simulator has a state variable called the simulation clock which holds the current system time and is incremented as the simulation runs. This variable is used for scheduling of the distinct discrete events from the future event list, which contains the set of discrete events currently generated that will occur at some (future) point in time.

There are many different types of simulation models that can be used. As previously discussed one of main differences is the treatment of time within the system, whether in a discrete or continuous fashion. Simulation models which treat time as a discrete entity can be furthered classified upon the different factors identified and used in modelling the world, such as activities, processes or events. In this project I am looking at specifically the discrete event modelling methodology for simulations.
2.1.3 Simulation models

Conducting a discrete event simulation exercise requires a simulation model that can be used as a method for representing a real world system, when constructing the simulator exercise. There are a number of different models that are widely used in the simulation world. Three of the most popular and widely used are petri nets, finite state machines and queue models. Each model has its own distinct advantages and disadvantages for using as a tool for system modelling in simulation exercises.

2.1.3.1 Petri Nets

Petri nets, also known as place/transition models, are “a graphical and mathematical modelling tool applicable to many systems”[3]. Petri nets were first introduced to the world in 1962 by Carl Adam Petri in his PhD paper “Kommunikation mit Automaten”[4]. Petri nets consist of places, transitions and arcs. Input arcs connect places with transitions and output arcs connect transitions with places. Each place can hold between zero and a specified number of tokens, the number of tokens in each place represents the current place state. Transition occur when a specified number of tokens are present in a transition’s input places. When a transition is fired a number of tokens are moved from the transition’s input places and into the output places. The exact distribution of the tokens depends on the arcs connecting the places and transitions. Petri nets can be “categorised as concurrent, distributed, parallel and nondeterministic”[3] and therefore are well suited to many modelling applications in the parallel simulation world.

2.1.3.2 Finite State Machines

A finite state machine or finite automaton “is a limited, mechanistic model of computation. Its main focus is the notion of state”[5]. Finite state machines consist of states and transition functions. A state stores information about the past, i.e. it reflects the input changes from the system start to the present moment. A transition indicates a state change and is described by a condition that would need to be fulfilled to enable the transition. Finite state machines can be represented by a state diagram, graphically describing the states and the transitions between them, or through a state transition table. A finite state machine can be deterministic (DFA), “if and only if, for each combination of state and input symbol, it has at most one transition”[5], or non-deterministic (NDFA), “if multiple transitions exist for the same combination of state and input symbol”[5].
2.1.3.3 Queueing Model

A queueing model’s main components are the customers and servers. A customer “is anything that arrives at a facility and expects to be serviced”[1]. A server “refers to any resources that provide an expected service for customers”[1]. When a customer arrives at a service two things can happen, if the service is busy the customer is put into a queue or if the service is currently idle, no customers are waiting, then that customer begins to be serviced. A queue has two main characteristics that affect the performance, arrival rate and the service rate. The arrival rate, \( \lambda \), dictates with what frequency customers arrive and the service rate, \( \mu \), dictates how long it takes for a customer to be serviced. These rates can be constant or include some element of pseudo-randomness, such as the Poisson distribution. Once one customer has finished being serviced the next one in the queue is then serviced, if no customers remain the queue becomes idle.

A queueing system can be described by “its calling population, the nature of the arrivals, the service mechanism, the system capacity and the queueing discipline.”[1]. The calling population of a system comprises of the total number of external available customer that could possibly enter the queueing system, decisions have to be made determining whether this population is finite or (approximately) infinite. The system capacity determines the limit on the number of available spaces in the model’s queues, an arriving unit will return to the calling population if there is no space to wait. Queueing behaviour indicates how the queueing size will affect the queue and in what order units will be removed from the queue for servicing.

Server utilisation, \( \rho \), “is defined as the proportion of time that a server is busy”[1] and can be determined by the following equation: \( \rho = \frac{\lambda}{\mu} \). If \( \rho > 1 \), where \( \lambda < \mu \), the server will always be busy and the queue will grow indefinitely if the calling population is infinite, which represents the total number of customer in the system. The size of queues in stable systems, where \( \rho <= 1 \), depends on the arrival and service process, whether constant or random. With Poisson distribution, for the arrival and
service rate, the queueing population will fluctuate in length but won’t grow indefinitely.

Figure 2.1 shows a M/M/1 queueing system. This type of queueing system is the simplest model and the M character symbolises Markovian arrival and service time density. This queueing model consists of a single server indicated by the last number. M/M/1 queueing systems assume a Poisson arrival process. This assumption is a very good approximation for the arrival process in real systems that meet the following rules[6]:

1. The number of customers in the system is very large.

2. Impact of a single customer on the performance of the system is very small, a single customer consumes a very small percentage of the system resources.

3. All customers are independent, their decision to use the system are independent of other users.

### 2.1.4 Parallel Simulation and causality errors

In many types of event simulation the traditional modelling paradigms operated in a sequential fashion and this caused enormous problems with regards to the execution speed of these models. Parallel simulation attempted to rectify this problem by extending the simulation models so they could be divided up and run in parallel to each other. This approach takes advantage of the natural parallelism inherent in many systems today. To construct a parallel simulation model the normal sequential model is taken and divided up into multiple separate sub models called Logical Processes.

A Logical Process (LP) contains several objects, the local virtual time, future event list, current event and messages[7], as shown in Figure 2.2. The future event list is a list of time stamped events which
are executed in order as the local virtual time increases. The individual LPs can pass time stamped messages between each other, such as exchanging events generated in their system that affect other LPs. With the LPs all acting independently of each other the internal state and virtual time of one LP may differ massively from another LP. This causes problems when a LP generates a time-stamped event for another LP which has already passed that point in time according to its own virtual clock. This kind problem of is called a causality error.

2.1.5 Dealing with causality error in parallel simulations

It is possible to ensure no causality errors occur within a parallel discrete event simulation if the system observes the local causality constraint, which is defined as “A discrete event simulation, consisting of LPs that interact exclusively by exchanging time stamped messages, obeys the local causality constraint if and only if each LPs executes events in non decreasing time stamp order.”[2]. Three different approaches have been developed to cope with this problem of causality, called conservative, optimistic and synchronous.

2.1.6 Conservative protocol

Conservative discrete event simulation obeys the constraint above to ensure that all LPs process messages only when it is safe to do so. LPs in this model “communicate with each other over separate links using non-decreasing time stamped messages”[8]. Each link has a clock associated with it that is set to the time stamp on the head message of the queue or the time stamp of the last message on the queue if it’s empty. Each LP selects the link with the lowest clock value and updates its local virtual time. If the queue is non-empty it processes the head message on this queue, sending out any messages along its LP links if generated. If the queue is empty then the LP blocks waiting for messages.

Whilst this approach guarantees that the causality constraints are satisfied, it incurs problems of its own, namely deadlock. Deadlock occurs “when a cycle of processes are blocked and each process is blocked due to another process in the cycle”[2]. To avoid deadlock null messages are sent between LPs that are connected in a round-robin fashion. Null messages are used to indicate a lower bound on all messages that will be sent in the future by that LP. The receiving LP knows that any future messages it receives, from the LP the message came from, will not have a time stamp lower than this and can proceed, instead of blocking, if appropriate. Whilst this approach is successful it can cause wasted bandwidth due to the amount of null messages being sent and also the overhead required to process
2.1.7 Optimistic protocol

The optimistic approach takes a rather more laissez-faire approach to the problem of causality. In this approach the direct synchronisation of all clocks in the system is ignored and they are allowed to proceed without notifying the others of their local times. Each incoming message to be processed has its time stamp checked against the local time and in the event of the local time being ahead a roll back mechanism is enforced. This process, called a Time Warp, resets the system to the state at the time of the time stamp on the message in the past[8]. Any messages that have been processed, in what is now the future, are resolved using anti-messages, which are sent out to the same LPs as these future events were. On receiving an anti-message a LP will itself perform a time warp. This process can lead to a cascade of roll backs throughout the system.

The disadvantage of this approach to causality is the increased memory needed for storing snapshots of the system at distinct time intervals to be used in roll back[8]. Large amounts of rollbacks can be extremely costly in terms of performance.

2.1.8 Synchronous protocol

A synchronous protocol can also be used to avoid local causality errors in parallel discrete event simulation. This protocol uses a global clock to enforce synchronisation between the distinct LPs and remove the opportunity for causality errors to occur. The global clock is calculated as the lowest of the minimum timestamps of all the logical processes running in the system. The global clock’s value is shared between all LPs who retrieve the global minimum and only process events which are lower or equal to this minimum value. After completing their event processing synchronisation takes place, where all the LPs inform the global clock of their minimum time stamp in order to recalculate the global minimum before the simulation continues on, repeating the previous steps.

Whilst the synchronous protocol is simple and easy to implement, it’s a very pessimistic approach and performance of the system is dependent on the cost of synchronisation for the global clock between all logical processes. In the worst case “the simulator may behave as a sequential one”[8], negating any performance benefits of parallelism.
2.2 Parallel Programming

2.2.1 Introduction

Computational power continues to grow according to Moore’s law[9], following the basic trend that available computing power will double every eighteen months. To this day, with some variation, this trend has remained true and permeated into other measures of computing capacity such as network bandwidth, various forms of memory storage etc, although with different time frames for the actual doubling. However as computing capacity grows, especially raw computational power, so does the omnipresent demand for more, as people try and attempt more ambitious and computationally intensive projects. To increase the availability of computational resources researchers have tried to look at other ways of providing the required resources, without resorting to simply trying to increase the performance of the raw components.

Parallel computing is described as “a set of processors that are able to work cooperatively to solve a computational problem.”[10]. That is the use of multiple central processing units that appear to work as one single processor in a normal way whilst executing programs. Using multiple processors together is not a new idea and was theorised about in the 1958 by Gill in his paper “Parallel Computing” in which he said, “There is nothing new in the idea of parallel programming, but its application to computers. The author cannot believe that there will be any insuperable difficulty in extending it to computers.”[11].

Since then parallel computing has evolved to become a very real approach to increasing collaborative computational power and there exists multiple methods to achieve viable parallel computing today. A parallel computing project today may involve not just multiple processors on one machine, but multiple machines all with any number of processors working together on one computational problem. To incorporate this new parallel approach the traditional computer architecture model has had to adapt in order to function correctly. Traditionally a computer system with one processor and memory store operated as a synchronous machine, reading data instructions from the memory and operating on these instructions, possibly modifying memory or internal registers. This type of architecture is called Single Instruction Single Data (SISD)[12]. This approach offers no real possibilities for parallelism although schemes such as pre-emptive multitasking may make it appear this way.
2.2.2 Parallel Architectures

A taxonomy for parallel computer architectures was first introduced by M.J. Flynn in 1972[13]. This attempted to categorise the wide spectrum of computers with parallel abilities into four main categories:

- **Single Instruction, Single Data (SISD)** As described previously, also known as a *uniprocessor* system.

- **Single Instruction, Multiple Data (SIMD)** This architecture meant the processors in a system would execute the same instruction stream on a different set of data in a “lock-step”[12] fashion, so that the program counter only proceeds once each step of that operation on each data set has completed. This approach is limited in what it is able to achieve and is only really feasible for specific problems.

- **Multiple Instruction, Single Data (MISD)** “A sequence of data is transmitted to a set of processors, each of which executes a different instruction sequence”[14].

- **Multiple Instruction, Multiple Data (MIMD)** This system “executes multiple instruction streams on multiple data streams simultaneously”[14]. Many multi-processor systems fall under this category.

Although Flynn’s taxonomy is still very useful the actual nature of parallel systems means that there is some crossover between categories. In the practise “the boundaries between Flynn’s SIMD and MIMD categories are blurred, few programmers could write 10,000 different programs for a machine with 10,000 processors”[15]. Instead many programmers write a single program, which can be replicated on each processor, each working on its own data with the combination of conditional statements to vary the execution path. This type of architecture is known as Single Program, Multiple Data (SPMD).

The most commonly used paradigm in parallel computing today is known as Multiple Program Multiple Data (MPMD)[10]. Multiple programs operate on separate data streams asynchronously to each other. Depending on the system set up, the data in memory that’s being operated upon maybe be either centralised in one place, or more likely, operating under a distributed manner, such that the program’s data will be located locally in memory to that processor.

Once a specific architecture has been chosen parallel computing still needs to be concerned with the method of splitting up computer code so that it can be operated on independently and, in the case of distributed memory, provide some kind of communication mechanism for synchronising returned results...
between processors. Message passing is one of the most widely used parallel programming approaches used today[16] and “in a message passing program, processes do not communicate through shared data structures; instead, they send and receive discrete messages”[15]. The message passing model has advantages over others, such as shared memory programming, for a variety of reasons including[16]:

- **Universality** The message passing model fits well on separate processors connected by a communication network.

- **Expressivity** Message passing has been found to be a useful and complete model in which to express parallel algorithms.

- **Ease of debugging** The message passing model, by controlling memory references more explicitly, makes it easier to locate erroneous memory read and writes.

- **Performance** Message passing provides a way for the programmer to explicitly associate specific data with processes and thus allow the compiler and cache-management hardware to function fully.

### 2.2.3 Message Passing Interface

The Message Passing Interface, MPI, was first defined in 1994 to “develop a widely used standard for writing message-passing programs”[17]. Since then the standard has undergone three revisions, currently standing at version 2.0. The popularity of the standard has meant a number of implementations of the standard have been developed and are widely used e.g. MPICH[18], LAN/MPI[19].

In its simplest form the standard defines a series of operations and interfaces that provide the capability to run multiple programs on different processors/machines and pass messages containing data between these separate processes, through collective or point-to-point communications. The standard has a very defined scope which leaves many decisions up to the implementations, such as process start up.

MPI’s set of functions provides more than simple send and receive primitives for unicasting single messages to each process. The notion of groups allows the partitioning of processes into separate logical collections. Each processes is identified in the group by a rank from 0 to n -1, where n is the number of processes in that group. Communication can be strictly limited between group members or between other groups, depending on the communicator used. Tags are also used to enforce ordering of
messages being passed between processes. Higher level functions such as the broadcast, gather and reduce, expand on the simple message passing to allow easy implementation of various collective parallel programming operations. Although the MPI standard defines over a hundred functions a much smaller number is usually need for most parallel programmes, in fact “a person can begin to write message passing programs with as little as six core functions”[16].

2.3 Grid Computing

2.3.1 Beginnings of the Grid

Since the beginning of the Internet, as the result of research into decentralised computer networks, the amount of interconnected computers has grown as an substantial rate. As a result of the “World Wide Web” the ever expanding Internet began to grow at an even greater exponential rate. By the late nineties the Internet was fast becoming a ubiquitous global network reaching into every walk of life. Whereas the “World Wide Web” was concerned with sharing of information, originally between scientists, computer researchers began to see another use for this worldwide computer network.

As bandwidth continued to increase the limitations of using computer systems geographically distant to each other fell away. Existing research into parallel computing began to look at ways of utilising the massive combined computational resources of computers connected via the Internet. This idea of sharing computational resources was the original motivation behind what was subsequently called Grid Computing. Computational resources is a blanket term for any services that a computer can provide, this includes processing capabilities, disk and memory storage, access to remote data sets or applications and much more.

The direction and aim of Grid computing has changed and mutated to incorporate new technologies that increase what is possible. Back in 1999 Grid computing was described as: “A computational Grid is a hardware and software infrastructure that provides dependable, consistent, pervasive, and inexpensive access to high-end computational capabilities”[20]. As Grid computing has evolved this definition has widened to “place less emphasis on specific computational resources, incorporating virtual organisations and collaborative resource sharing”[21].

The term Grid Computing draws parallels with traditional utilities companies (water, electricity, gas) and the view of computing resources as a service, that can be turned on or off arbitrarily as demand requires, much like a tap. The evolution of Grid computing can be seen as three distinct stages up to
the current day. The first efforts at projects, resembling what would become Grid computing, were aimed at linking together remote computer locations and was termed *meta computing*. Projects such as FAFNER[22] and I-WAY[23] were some of the first distributed computing projects taking advantage of geographically remote resources to collaborate towards a common goal in such a way. FAFNER was concerned with the computationally intensive factoring of large numbers relating to cryptography and I-WAY was built to link existing supercomputers over existing networking infrastructure using the then new ATM network technology.

### 2.3.2 Evolution of the Grid

The second generation of Grid computing is a lot closer to the current model and began with the first working Grid that came out of a project at Argonne National Laboratory in September 1997, titled “Building a Computational Grid”, which was followed a year later by the publication of Ian Foster and Carl Kesselman’s book *The Grid*[24].

The second stage of Grid computing attempted to broaden the scope beyond specific tasks such as linking existing supercomputers through the tackling the three distinct issues of heterogeneity, scalability and adaptability[25]. Heterogeneity refers to the different computing platforms that could be used simultaneously in a Grid system which must appear as one homogeneous resource provider overall. Scalability allows for the Grid to grow in size as resources are increased by adding more machines in various different locations. The distributed and varied nature of a Grid system means resource failure will be a common occurrence and the systems between, used to co-ordinate Grid infrastructure, must be able to deal with this in a manner transparent to the overall workings.

### 2.3.3 Virtual organisations

As Grid computing began to solve many of the problems which it first encountered when originally trying to coordinate the use of heterogeneous, geographically separate resources, the search turned to using Grid computing more effectively than just utilising the large-scale computational power it provided. In many aspects Grid computing has followed closely the ever changing nature of the World Wide Web, from the initial problems of wide area communication networks, and so as the web started to change into a more service orientated platform so the Grid environment followed.

From this approach the idea of “virtual organisations” manifested itself. The term virtual organisations was first discussed in “The Anatomy of the Grid: Enabling Scalable Virtual Organisations”[26]
which attempted to look at the current state of Grid computing and redefine the “Grid problem”. A virtual organisation can be described as “a set of individuals and/or institutions that, as part of a common goal, participate in dynamic, collaborative, coordinated resource sharing and problem solving with highly controlled definitions of what is shared and by whom”[26]. They identified problems that virtual organisations might face using existing distributed technologies to function and attempted to set out an architecture in which Grid computing could be used for this purpose.

### 2.3.4 Grid Architecture

Enabling widespread adoption of Grid computing through “virtual organisations” meant the standardisation of a common Grid architecture, which would be used when building the technologies needed to facilitate the dynamic, coordinated resource sharing associated with VOs. A standard Grid architecture would most importantly promote interoperability between all Grid components that followed the agreed architecture model because “effective VO operation requires that we be able to establish sharing relationships among any potential participants. Interoperability is thus the central issue to be addressed.”[26].

![Grid architecture model](image)

The now standard Grid architecture is shown in figure 2.3, next to a diagram of another standard architecture widely known as the Internet protocol. The use of a layered architecture model, like the approach now employed with the Grid’s, has been around for a long time and was widely used in the OSI (Open Systems Interconnection) reference model, that was created in order to enable interoperability for network communications. This separation of the model into layers increases interoperability because each layer can only interface with other layers directly above or below through a series of abstract
interfaces, meaning that the actual implementations at certain layers can change without affecting the overall system.

The Grid architecture diagram in figure 2.3 shows that the actual model closely resembles an hourglass shape, with the bottom and top layers being the widest whilst the middle layers are narrower. This is an intentional design choice which follows the principles of the hourglass model, “the narrow neck of the hourglass defines a small set of core abstractions and protocols (e.g., TCP and HTTP in the Internet), onto which many different high-level behaviours can be mapped, and which themselves can be mapped onto many different underlying technologies (the base of the hourglass)”[26]. In the Grid architecture model the middle layers, known as the neck “consists of Resource and Connectivity protocols, which facilitate the sharing of individual resources. Protocols at these layers are designed so that they can be implemented on top of a diverse range of resource types, defined at the Fabric layer” and this allows “a wide range of global services and application-specific behaviours at the Collective layer”[26].

2.3.4.1 Fabric Layer

The bottom layer on the model represents the interface to computational resources that are to be shared on the Grid and does this by providing a set of operations that can be performed for each resource. A computational resource is a broad description that encompasses anything that could be shared on the Grid from storage devices to collective processing capabilities and beyond. Fabric components “implement the local, resource-specific operations that occur on specific resources (whether physical or logical) as a result of sharing operations at higher levels”[26]. The complexity of the implementation at this level will dictate the simplicity of deploying a resource onto the Grid but also influence the amount of work higher levels will have to take on.

2.3.4.2 Connectivity Layer

This layer sits on top of the fabric layer and defines a core set of protocols for connectivity over a network and the related security issues such as authentication. The connectivity requirements dictate the features of any computer networking protocols that are to be used for communication in any Grid. These specifications draw heavily from already existing network protocols in wide use such as TCP/IP and the requirements of these protocols when used in a wide area network such as routing, naming etc. The inherently dynamic and diverse nature of Grid computing means that security is an extremely important aspect, especially the restrictions on the availability of resources. Authentication for resources,
delegation of trust in the system and integration with existing security solutions is defined in this layer.

2.3.4.3 Resource Layer

Building on top of the capabilities provided by the connectivity layer and the resources at the fabric layer, the resource layer provides for the “secure negotiation, initiation, monitoring, control, accounting, and payment of sharing operations on individual resources”[26]. At this layer all the operations are still concerned with individual resources, operations on multiple resources are resolved in the next layer. Protocols at this level can be broken down into two classes, information protocols and management protocols. Information protocols are there to provide information regarding the state and structure of a resource. Management protocols are concerned with the negotiating of access for a given resource.

2.3.4.4 Collective Layer

The collective layer is called so “because it involves the coordinated use of multiple resources”[26]. Information about multiple resources is collected and manipulated here through the use of multiple operations at the lower, single resource, layer directly below it. This layer is a wider layer because a lot of the collective operations needed for real Grid applications are performed here, including resource brokering used in many of the Grid middlewares, and it builds upon the narrower layers below. This layer usually provides a series of APIs and protocols for use by the application layer above. Operations at this level can be tailored to meet the demands of specific VOs or more generalised for use in a number of applications.

2.3.4.5 Application Layer

The top and final layer in the Grid architecture model is the application layer. This layer builds on top of all the previous layers, using the protocols and APIs provided by the collective layer below to build applications utilising a whole range of services and resources that are provided in a virtual organisation. Provisions are also provided for this layer to communicate directly with some of the other layers, resource and connectivity, making this layer extremely flexible.

2.3.5 Grid middleware

To overcome the three obstacles of heterogeneity, scalability and adaptability, the Grid community has extensively used middleware to enable Grid computing to function. Middleware is generally considered
to be the layer of software sandwiched between the operating system and applications, providing a variety of services required by an application[25]. The required middleware is present on every resource in the Grid and acts as a homogeneous environment, shielding away the varied nature of underlying resources on the Grid. Middleware is also used to deal with many of the important issues that are required for a Grid to work effectively, such as resource discovery, security authentication and job control. One of the most popular middlewares available is the Globus Toolkit[27]. Globus first originated from the Grid project at the National Laboratory in 1997. Since the first release in 1997 it has grown in popular and functionality, currently on version 4.0 encompassing the majority of components needed in a modern Grid environment.

2.3.5.1 Globus Toolkit

The Globus toolkit[27], displayed in figure 2.4, consists of a layered, modular architecture split into five distinct areas of security, data management, execution management, information services and common runtime components[28]. This approach allows for only a small number of essential, core components which then can be used to build up more abstract and advanced features of the toolkit. Secondly, its modularity allows the use of one section of the toolkit without necessarily needing other sections. The Globus toolkit also provides a large and well documented API allowing the higher level implementation details of Grid applications to be left to the user. Adopting the open source paradigm and using standards
extensively has helped the Globus toolkit become one of the ‘de facto’ standards for computational Grids.

Globus is not the only Grid middleware available today and other examples such as Legion[29] and Unicore[30] are also popular. Legion differs from the Globus toolkit in that it uses the object-orientated methodology in its approach, encapsulating all the different aspects of a Grid as classes, forcing the end user to adopt this approach when building upon the Legion toolkit. Another popular middleware in Grid computing is called Unicore. Unicore, which attempts to provide a uniform work environment for the user, dividing the architecture into just three layers, the user, server and target system.

2.3.6 Grid Services Architecture

The Open Grid Services Architecture (OGSA) is a standardised framework to “support the creation, maintenance, and application of ensembles of services maintained by virtual organisations”[25]. The OGSA represents a convergence of traditional web services with the Grid computing platform and was introduced in a paper at the Global Grid Forum in 2002[31]. It provides standard interfaces for virtual organisations to facilitate discovery of their resources on a Grid, dynamic service creation mechanisms, event notification and so on. OGSA uses many of the protocols associated with traditional web services such as the web services description language for service discovery. The previously defined Grid architecture was concerned with the protocols required to enable interoperability between Grid components whereas this version of the Grid architecture, building upon the previous work, is focused on “the nature of the services that respond to protocol messages”[31]. This enhancement of the Grid computing vision leads to a situation where “a Grid is an extensible set of Grid services that may be aggregated in various ways to meet the needs of VOs”[31]. Traditional web services technologies, that are becoming common place on the Internet, are being adapted and evolved for use on the Grid platform. Taking the already available tools and libraries, used in normal web services operations such as SOAP, WSDL, resource description and discovery, and implementing these tools for the Grid, to enable the creation of the Grid services paradigm.

2.3.7 Parallel Programming and the Grid

As Grid computing has emerged as a system of geographical disperse, heterogeneous systems, moves have been made to try and extend traditional parallel programming models to utilise this new source of enormous computing resources. MPICH-G was the result of an attempt “extends the Argonne MPICH
implementation of MPI to use services provided by the Globus Grid toolkit.”[20]. This result of this effort was described in a paper in 2000 by Foster and Karonis[20].

Although Grid computing became more commonplace throughout the nineties, building functional Grid applications still remained problematic and needed the programmer to learn the intricacies of whole new categories of systems. Traditional parallel programming using message passing via the MPI library was more widely used and understood however. MPICH was used as the building block for creating a Grid compatible version of the MPI library. MPICH was the a commonly used implementation of the MPI standard and also open source.

The wide area distribution of computing resources involved in Grid computing gave rise to its own series of problems which weren’t usually relevant when using traditional MPI, as the processes were usually located on geographically close computer systems. These problems included authorisation, authentication and the differing topologies of underlying networks amongst others[20]. MPICH and Globus were combined together to provide the user with the expected homogeneous computing environment associated with MPI programming by screening away the underlying Grid technologies involved. Normal components associated with Grid computing such as authentication and data transfer were dealt with transparently in the library, so they would appear hidden to the end user. The same methods for compiling and running MPI programs was incorporated for seamless integration of parallel programs on a Grid. A follow up paper was then publish in 2002 by Foster, Karonis and Toonen[32] which documented the results of seeking to expand the current version of MPICH-G to MPICH-G2, extending the MPI library functionality and addressing problems that had been apparent with the first version of the software.

2.3.8 Examples Grids and their Applications

Despite being a recent and emerging technology, having only been conceived in the past fifteen years, Grid computing is now prevalent in many specific areas, such as scientific research, having a large and important role to play. To give an idea of the types of work Grid computing is now involved in a brief overview of a sample large scale Grid and an application which utilises it is given below.

2.3.8.1 White Rose Grid

The White Rose Grid[33] is a large scale, computational Grid used to connect together resources located at the Universities of Leeds, Sheffield and York via a high speed wide area network connection for
researchers engaging in e-science activities at those universities. It was launched in the summer of 2002 and is comprised of five separate nodes, three at Leeds and one each at York and Sheffield, of high performance computing clusters connected over the YHMAN (Yorkshire and Humberside Metropolitan Area Network), enabling researchers access to a range of computing resources which best suit their needs. The computing nodes and topology of the White Rose Grid is shown in figure 2.5 (note: this diagram doesn’t include the new third Leeds node).

2.3.8.2 DAME

The Distributed Aircraft Maintenance Environment (DAME) “is an e-Science pilot project, demonstrating the use of the GRID to implement a distributed decision support system for deployment in maintenance applications and environments”[34]. In association with Rolls-Royce this project was created to investigate the applicability of the Grid services paradigm to the development of systems for the diagnosis and maintenance of aircrafts, which need to use large data sets situated in remote locations with local decision support systems. This project used the existing White Rose Grid infrastructure as a Grid test bed for developing the technologies involved.
2.4 Summary

Grid computing is an extremely new technology that, although still in its relative infancy, is starting to show real promise as an exciting breakthrough in providing groups of users with the ability to pool their resources in order to achieve common goals, regardless of their physical locality. The world of computer simulations is an area which already has an enormous use for large scale computational resources, especially processing power, having traditionally used large groups of computer clusters in order to accomplish the complex simulation exercises. Running simulations in parallel, on a group of computers in a cluster, requires some parallel programming construct to enable communication between different parts of the simulation. One of the most popular parallel programming paradigms, message passing, has a number of libraries available today to accomplish this. A widely used implementation, MPICH, has released a Grid enabled version that interacts with a specific Grid middleware, the Globus toolkit, to enable normal parallel programs to run transparently on a Grid. With the parallel programming technology now Grid enabled running computational simulations on a Grid has never been easier or more accessible, yet it remains to be proven whether the enormous change in environment, that comes with Grid computing, is a viable place for parallel discrete event simulations.
Chapter 3

Design

3.1 Introduction

In this section the overall design of the various aspects of the project will be covered in detail. Covering the software design methodologies and initial class design, laboratory test environment and finally the performance metrics and experiment proposals.

To evaluate the performance of parallel discrete event simulation on a Grid test bed I need to set out clearly designed experiments that can be used to test my criteria. This involves being critically aware of the environment I'm conducting experiments in and the pitfalls associated with testing performance of distributed, parallel systems. One of the most important aspects of any performance test is that the results are ‘reproducible’[35], without this factor any data generated is inherently unreliable and any conclusions drawn could be false. Clear objectives need to be produced that the experiments can be evaluated against, reducing any possible ambiguity to a minimum. These tests may either build upon existing frameworks available or create new ones depending on the requirements.

The software for the Parallel Discrete Event Simulator also needs to have a thorough design before the implementation can begin. Without a sound design behind the project it could easily fail to behave as expected, and it might not fully reflect a real world system. Understanding of the various technologies involved is needed to create a software design that will behave as expected and be as close a model as possible to the simulated system. The same goes for the laboratory environment, which is an extremely important factor in designing the experiments and software. The underlying hardware and software used in the laboratory will have an enormous impact on the simulation and needs to be accounted for in both the software and experiment design.
3.2 Environment

The initial parallel computing and Grid test bed will consist of two machines, iri01.leeds.ac.uk and iri03.leeds.ac.uk. These machines are high power multiple processor machines that are currently used in the University for traditional, large scale parallel processing applications and have a test bed Grid enabled on both machines.

3.2.1 Software

MPI (Message Passing Interface) was a standard first defined in 1994 to “develop a widely used standard for writing message-passing programs”[17]. Shortly after this standard was published implementations began to appear for practical use. Various different implementations of the MPI standard, currently at version 2.0, are available and contain different parts of the standard.

One of the most popular implementations of the MPI standard is MPICH[18], designed and maintained by the mathematics and computer science division at Argonne National Laboratory. This implementation conforms to the newest version of the MPI standard and has been around since the first conception of MPI. This combination of completeness and maturity makes MPICH an excellent choice to use as the MPI implementation in my experiments. Secondly MPICH is an open source research project that publishes papers detailing the implementation details of the project. This allows me to fully understand the middleware architecture and use this information when designing the experiments. Another feature of this particular MPI implementation is that a Grid-enabled version of the MPI standard is already available[36] from the same developers. The experiments will be conducted using version 1.2.7 of the MPICH and MPICH-G2 software.

Grid middleware is used to facilitate interactions between process in virtual organisations. The middleware hides the heterogeneity inherent in Grid technology and provides services for resource allocation and security amongst others. The Globus toolkit[27] is a software service containing toolkits and libraries to facilitate develop of Grid applications. Rapid development of this open source project allowed it to expand from an initial release in 1998 to version 4.0 by 2005. In my experiments I will be using Globus v4.0.1 on iri03 and Globus v2.4.3 on iri01. The Globus toolkit is a well matured and stable Grid middleware.
3.2.2 Hardware

Both machines that are going to be used to run the simulation experiments are SGI Onyx machines with different operating systems and hardware specifications.

*iri01.leeds.ac.uk* has twelve 400Mhz (MIPS R12000) processors, 6144 Mbytes of RAM, Gigabit Ethernet networking interface, SCSI disk and NFS mounted storage and runs the IRIX64 6.5 operating system.

*iri03.leeds.ac.uk* has four IA-64 (Itanium 2) 1.5 Ghz processors, 5834 Mbytes of RAM, 100Mbit Ethernet networking interface, SCSI disk and NFS mounted storage and runs the SUSE LINUX Enterprise Server 9 (ia64) SMP operating system.

The machines are connected using a series of linked switches over a local area network to account for the different network connections, 100Mbit and 1000Mbit.

3.3 Measuring performance

Measuring performance in any situation requires a suitable metric by which to gather data and compare results on. Any experiment concerning performance requires a clear value which dictates how the experiment will be designed. In the experiments I’ll be looking to evaluate the performance of PDES on the Grid. To enable myself to do this I must define what is meant when talking about *performance* and explain the metric by which I’ll be measuring the performance of my system. I also need to anticipate the sources of error than can creep into the system when attempting to measure performance of parallel systems. By anticipating these sources of error I can attempt to remove or work around their effects in the experiments.

3.3.1 Performance metrics

A metric is a standard of measurement and allows the contrasting of results from the experiments against each other. A variety of different metrics are available when measuring performance of systems in the tests and choosing the right one is crucial, to correctly compare PDES on the Grid with traditional parallel programming platforms. Here I look at some of the metrics appropriate for the experiments to use when measuring performance.
3.3.1.1 Time

Using time is an obvious metric for measuring performance in a parallel system. However if time’s used as the performance indicator it leaves me with outstanding questions such as where and how in the experiment should time be measured? I must take into account a number of extenuating factors that will affect the overall time taken for a parallel simulation program to execute. The different test environments I’m comparing will mean that I have to use my knowledge of the protocols and systems to account for the multitude of external influences when measuring time.

Initialisation and shutdown time in the two setups may vary widely, dependent on external factors, but these delays happen when the programs aren’t performing any relevant computational activity and so obviously this needs to be accounted for in the overall results. The time needed to distribute and run the various processes may be far higher on the Grid due to initial start up costs however with most parallel programs running for large periods of time, these initial costs become irrelevant to the overall performance measurement of the system. The incurred times might also vary widely between the tests, for reasons such as machine load or resource allocation on the Grid, and so they can’t be incorporated into the final results. Another factor is what I actually record when calculating time, the complete actual spent in the simulation by the program or just the CPU time on the controlling host computer. Provided that the factors mentioned above can be mitigated or minimised, time provides an excellent measure of the overall performance of a system.

3.3.1.2 Data transfer

Another visible metric that could be used when measuring performance of parallel simulation is data transfer. Using data transfer means again defining what I perceive as ‘data transfer’ and what is relevant in the experiments. Data transfer is used here to describe the general scenario when bits of data are sent over a network between the various computers running a parallel simulation. In parallel simulation there will be two main activities that involve large scale data transfer, firstly the initial procedure for distributing processes between the computers involved in the parallel simulation and starting these processes on the hosts. Secondly the data transfer involved in the actual message passing between separate physical computers participating in the parallel computation. The relative cost of each of these procedures will be application dependent.

Measuring data transfer of the PDES exercise and will give a good indication of the data overheads
associated with MPI over Grid compared with traditional MPI. Grid based simulation would expectedly generate large data transfers due to the underlying Grid based network traffic involved. This would include the initial resource allocation and subsequently load balancing that might be involved using MPI over Grid services. The measurement would allow an evaluation to discover the order of the data transfer increase and see how it scaled.

3.3.1.3 Load

A final indicator of performance may be measured using the system load during Grid based MPI and traditional MPI simulation. Load is a vague term and is also known as utilisation. Measuring the load of a system may entail measuring the CPU, memory, disk or network utilisation throughout the simulation period. Measuring the CPU, memory or disk load would show any performance impacts of each of the simulation approaches on the machines involved. This might include measuring the load at each individual computer involved or just the controlling host.

The single most important point to consider when measuring network load is the bandwidth for each host, the overall total and current usage. Principally the total bandwidth will dictate how much data a parallel simulation host will be able to send and receive data whilst the current usage will have an enormous impact on the network load. External programs utilising large portions of the bandwidth at the same time as the simulation will make it appear that the network is far slower than it actually is.

3.3.2 Sources of error

Any experiment that doesn’t consider the external influences of the test environment will run the risk of failing to generate accurate and reproducible results. Sources of error can result from “the Unix operating system, together with network hardware and software, that introduce sporadic intrusions into the test environment”[35]. By cataloguing and subsequently addressing the potential sources of error in the experiment, in the design stage, I can seek to minimise their impact on the test results.

Here is a list of some of the most important sources of error in the experiment and how I’ll be dealing with them.

**Measuring time between incorrect intervals** When using time, as a measure of performance on the system, the experiment must be exact about what exactly it wants to record. Without a firm understanding of the processes involved and a clear definition of the simulation performance, the
simulation tests may actually record time associated with factors that aren’t included in the definition of performance, such as the initial process distribution. In the experiment I’ve defined precisely what I mean by performance and what I shall be recording in the timing of the parallel simulation. By having definite start and end points I can tailor the timing mechanism to only include this interval.

**Resolution of clock** Depending on the granularity of the experiment the resolution of the clock mechanism, which might be dependent on the host computer’s load, might have a large impact on the result set. For the experiment granularity of milliseconds is small enough whilst the possible permutation associated with the system clock resolution is in the order of microseconds, if not smaller. Therefore this factor won’t have an realistic impact on the results.

**Ignoring the test environment condition** The test environment that the experiments will be running in is open to other users for parallel simulation exercises and I have no facility to restrict access during the tests. The system load, consisting of CPU load, memory and disk usage and network utilisation, can have an enormous impact on the ability of the tests to perform correctly. “Ignoring contention with unrelated applications or jobs”[35] will cause unexpected tests results and won’t reflect the actual performance of the parallel simulation exercise. Using a collection of operating system specific commands will allow me to check the system load components and determine whether it is an appropriate time to run the test beforehand.

**Inappropriate experiment values** Using inappropriate experimental values will make any evaluation on the results returned susceptible to misinterpretation. Running a simulation exercise that entails a great number of individual processes may result in the performance being dependant on the simulation hosts’ processing power and fail to accurately reflect the Grid based performance change. Similarly the total message count must be large enough so that any performance impact can actually be reflected in the data, using small numbers of total messages might mean the results are more susceptible to environmental influences. Initial preliminary tests, such as maximum throughput and simulation load, will allow me to decide on the values in the actual experiments.

**Topology setup** One of the most important factors to consider is the distribution and communication between the separate LPs in the simulation. How the LPs are distributed and in what order they communicate will have an enormous performance impact on the experiment’s results. If the majority of the communication is between LPs that are situated together on the same host then
the actual message passing over the network will be reduced to a minimum and this obviously
wouldn’t be an accurate reflection of a real world situation. To enforce message passing only
between processes on separate hosts, so that all the messages pass over the network, the topology
of the LPs communication links needs to be carefully setup.

**Miscellaneous errors** A whole number of unforeseen errors caused by previously undetected sources
could have an impact on the test results. Whilst it is possible to remove or minimise a large
proportion of the errors involved, it is never possible to completely remove everything that could
interfere with our simulation. To counter this problem the simulation experiment needs to be run
for a sufficiently large sample rate. With a suitable sample rate the data can then be analysed
using traditional statistical methods to negate the effect of any anomalies in the data set.

### 3.4 Proposal

**3.4.1 Performance metric**

In the experiments I’ll be principally concerned with evaluating the performance of the simulation sys-
tem using the metric of time. Provided the experiments are designed with appreciation for the possible
problems, associated with measuring time in parallel simulation, I believe that this specific metric will
give the most accurate results with which to evaluate the platforms. Furthermore the experiment will be
related to capturing the overall time spent in simulation, from start to finish, rather than just the CPU
time for the simulation. Restricting measurement to CPU time would not give a correct representation
of the possible overheads that could occur when using Grid services. It’s the external Grid procedures
that are most likely to incur performance problems. Also, measuring CPU time would also lead me to
having to decide what I wanted to consider in the overall CPU time, the single host computer or all the
host simulators involved. The results would also be enormously dependent on the processing power of
the computers involved in the simulation.

Assuming I’ve correctly resolved the start and end points of the simulation program, that contain the
relevant portions that I want to measure, using the overall simulation time is an excellent way to judge
the performance of a system. Measuring the simulation systems in this way will give me a complete
overview of how the entire simulation exercise works. It’ll be directly affected by the introduction of
MPI on the Grid, as it’s measured at a high enough abstraction in order to involve multiple aspects of
a system that might be affected. The computational power needed for Grid computing, the extra data
transfer, increased memory requirement and any Grid communication procedures involved will all have an impact on the overall time used in the simulation and be reflected in the results. Using a less general metric, such as CPU load, could fail to reflect a number of elements involved in the performance of parallel simulation. For example, only measuring network load would fail to appreciate the extra computational power needed for Grid computing services on each of the simulation computers. The broad category of factors that this measurement will account for means that extra care and diligence will have to be taken when designing the experiment, as any one of these factors could have an adverse and irregular impact on the results. Taking into account the possible problems described in the previous section, coupled with an appropriate number of repeat tests being run, will allow me to spot any anomalies in the results.

### 3.4.2 Experiment proposal

The experiments will use two different variables during the tests attempting to measure performance, messages generated and message size. The main variable used will be the number of messages generated and timing how long the system takes to complete transfer of this pre-specified number of messages. An initial value of 5000 messages is large enough to be able to gain an appreciation of any performance impacts and each subsequent step of the experiment will increment the message count by 5000 up to a maximum of 50,000. A range of ten liner increments will allow enough scope to interpret the results, spot any patterns of scaling and is of a large enough variance to realistically have an affect on the overall time. Each experiment will be repeated to retrieve a data sample of 3 results and these results will be statistically analysed, using standard deviation and mean, to account for any anomalies in the data and look for overall patterns.

Secondly, the experiment will also be varying the message data size, to test how this factor affects the performance. The initial data size will be a small value, 100 bytes, to simulate systems where there is minimal data transfer. The following message sizes will increase by an order of ten, up to a maximum of 100,000 bytes, allowing simulation of systems where the message content is relatively large. As communication software becomes more sophisticated, simple tests become less indicative or the performance of a full application[35] and so by using two separate variable quantities in the experiment it allows the tests to simulate an enormous range of real world scenarios that PDES systems are used in, whether they send large amounts of messages with small amounts of data, small amounts of messages with large data content or any combination of the two. This experiment will allow me to
test the performance of the system on the Grid comparatively with traditional MPI simulation in a wide manner of different scenarios, helping to narrow down the factors directly involved in the performance of parallel simulation on the Grid.

### 3.4.3 Topology

The queueing simulation will consist of a series of distinct LPs, passing messages between each other via their predetermined input and output links. As the LPs will receive and send messages in a specific pattern the layout and topology of the queueing network needs to be thoroughly considered, along with the distribution and amount of LPs in the simulation.

Using an excessively large number of LPs could cause the simulation hosts to struggle to cope with the demand. This performance impact might be falsely interpreted in the results as being attributed to MPI on the Grid. Secondly the experiments I’m conducting are designed to test the overall performance of parallel simulations and as such the amount of actual message passing over the network, between logical processes during the experiments, needs to be kept at a maximum. With each LP sending and receiving messages only through their specified links, by splitting the total number of LPs over the two hosts and then enforcing a serial queue topology, I can enforce a policy that every message must travel over the network.

The two machines involved in the simulation have twelve and four processors respectively and ideally the systems will be using the individual CPUs for a single LP, to maximise performance. Creating a serial queue topology therefore requires no more than four LPs on the machine with four processors. Therefore because all LPs in the system should only be sending messages to LPs on the opposite host, I can only have five LPs on the host with twelve processors. Using nine LPs in the queueing simulation allows the experiment to have enough realism whilst not overloading both simulation host computers, figure 3.1 shows an overview of the topology in the experiment.

The actual distribution of the process ranks to computers is irrelevant assuming that the processes are split accordingly over the two hosts, five and four. I can determine which processes are on each host and set their input and output channels accordingly to reflect the actual topology. In the experiment the LP with rank 0 will act as a generator, spawning the message arrivals in the system according to the arrival rate, with no receiving channels from other LPs. The LP with rank 8 will have no output links and upon arrival of a message, from its input links, it’ll print out the message content contained. All the other LPs in the system will act according to the normal conservative PDES rules, processing messages
3.5 Software Design

3.5.1 Methodology

The developing of the software components for this project will follow a particular methodology for software engineering, to give an overall structure and order to the process. A number of different methodologies exist and a brief description of some are given below.

**Prototyping** This methodology employs the paradigm of rapid prototypes, working models of the system, which are built quickly and then refined in further versions until the design objectives are reached and a suitable application has been developed. The results of each prototype are tested and the feedback generated is feed back into the design model to improve the next software prototype.

**Waterfall** This model for software development is the oldest and most widely known. In this model the development is broken up into separate stages, usually requirements, design, construction, integration, testing, installation and maintenance. Each stage must be completely finished before the process can move on to the next one. This process can be very brittle if problems occur at a later stage.
**Test Driven Development** This process involves writing a series of test cases that are used to verify the correctness of the software before any actual implementation of the software is completed. Once a large series of test cases have been developed, that test each part of the system for correct functionality, the implementation of the software can began. This process is considered complete when all the test cases are passed.

During the implementation stage I will be using the prototyping methodology as part of the development process. This will involve rapid prototyping of my system combined with a series of development milestones that are to be reached with each prototype. I chose this methodology due to my familiarity with it during my time at university and it was the process I followed whilst on my placement year. The use of rapid prototyping will help me to learn MPI and Grid technologies through a gradual process, as I build successive prototypes of varying complexity, without trying to tackle the entire project at once. An initial Gantt chart was developed, Figure 6.2.6 in Appendix C, that incorporated a series of milestones that would allow me to measure the development progress as I built successive prototypes.

### 3.5.2 Design diagrams

Throughout my time at university and on my industrial placement I used the Unified Modelling Language (UML) to create the design models and specifications used in the software development process. It is therefore the natural choice for use in the initial design stage of any software development in my project. Using UML allows the system to have an overall design specification, detailing the program logic and outline, before any implementation begins.

An initial class diagram, Figure 6.8 in Appendix D, was completed for the system showing two classes that will make up the application along with all the details about their methods and members and the relationships between the classes.

As well as the initial class diagrams I designed an sequence diagram, Figure 6.9 in Appendix D, describing the internal behaviour of the system, represented by the two objects and their lifelines. This diagram shows the main software’s logical flow for each process involved in the simulation exercise.

### 3.5.3 Parallel Discrete Event Simulator

A PDES implementation acts as a mechanism on top on of which simulation applications can be built, by providing the underlying activities of PDES such as transfer events between LPs, event queues and
conforming to the input and output waiting rules. PDES can be implemented in three different forms, conservative, optimistic and synchronous as discussed in the previous chapter. The conservative protocol is simpler than the optimistic protocol to implement but is less pessimistic than the synchronous protocol, which can perform like a serial simulator in the worst case.

The simulation experiments I will be performing will use conservative PDES because of the obvious time constraints which limit the scope of the project. Conservative PDES is the most appropriate protocol to implement considering the project scope and my background, having had no previous experience with PDES, parallel programming or Grid computing.

3.5.4 Simulation exercise and model

For the purpose of my experiments I have chosen to simulate a queueing system using the queueing model outlined above. This system will consist of a number of separate queues connected together, passing customers between each other, who are eventually serviced by the server. Queueing systems are widely used in the simulation world and the characteristics of the queueing model fit well with the architecture of LPs in PDES, allowing easy mapping of the queues in the experiment to the individual LPs involved.

The arrival and service rates in my system in the experiment will be equal, to give a system utilisation of $\rho$ at most times. The actual customer arrivals and service times will be generated using the Poisson distribution mechanism to introduce an element of randomness into the system, whilst the service and arrival rates being equal will keep the system stable i.e. no queue will grow indefinitely. With a very low value for $\rho$ any performance impact that occurs with Grid computing could be hidden during large idle periods, i.e. the system could ‘catch up’. The arrival time mustn’t be set too high as the system would then be relying on the processing power of the host systems to generate messages as fast as possible. This would skew the tests in favour of the host system with the most processing power available, giving undue importance to externalities such as system load at the time of the experiment, leading to incorrect results.

The calling population, total number of customer in the system, and system capacity, total number of customers allowed to queue at once, will be infinite in my experiments, following the standard M/M/1 queueing model.
Chapter 4

Implementation

In this chapter, following on from the initial design stage, I will talk about the implementation of the project. Starting with the practical details concerned with the software and then through to the specific implementation, detailing how the application side of the project is built. The details of how I’m going to run the experiment and capture data for analysis will also be explored here. Problems that occurred during the implementation will also be detailed, along with the solutions, including any changes to the initial design or experiment.

4.1 MPI

This sections details the implementation specific decisions with regards to the MPI standard and how they affect the project.

4.1.1 Send and Receive Protocols

MPI provides both point-to-point and broadcast communications methods. Communication is controlled by membership of predefined groups and can either send messages from one process to another, or from one process to the rest of the group. Data can also be ‘gathered’ in both ways, from one process or all. Point-to-point communication was excellently suited for use in a queueing simulator because each queue passes events directly from one queue into another, after being serviced, and this could be directly achieved using this type of communication. Using point-to-point communication makes the idea of individual group creation redundant and so this isn’t something I needed to be concerned with. All processes being executed are automatically members of the MPI_COMM_WORLD group, which
can be used as the communicator handle. Separate individual groups could have been setup for each communication channel but this was made redundant by the usage of point-to-point communication coupled with the automatic creation and handling of message send/receives.

Sending messages between individual MPI processes requires careful consideration of how the implementation will carry out the actual transfer of data involved. Point-to-point communication in MPI uses direct Send and Receive method calls, similar in semantics to direct Unix sockets communication. Each call specifies the source/destination of the message, the group communicator, message length and also uses a tag field which can be used to indicate a message's purpose. “When MPI sends data the above information forms an envelope that is used in the matching of send and receive calls. Sending a message consists of sending the envelope followed by the data”[16]

The MPICH implementation provides a number of different MPI Receive and Send methods that differ upon the underlying algorithm used for data transfer. These methods fall into two main camps, blocking or non-blocking communication. Blocking communication methods mean the calling process will be “blocked” from any further activity until the method being called has completed its network communication, whilst non-blocking allows the process to proceed once the network communication has reached a specific point, data has been copied to client side buffer or a receive call has begun but not completed. MPI provides a variety of different send and receive methods that differ on their underlying networking communication procedures.

For the experiment I will be using the default MPI_Send and MPI_Receive methods to send messages between the processes. Both of these methods are classified as blocking because they involve a degree of waiting before each call returns. In MPICH the default protocol is called “eager” and in this protocol, “data is sent to the destination immediately. If the destination is not expecting the data (e.g., no MPI_Recv has yet been issued for it), the receiver must allocate some space to store the data locally.”[37]. The receive call blocks explicitly if there is no data waiting to be received either from another process’ send call or from the local buffer. The send call blocks until all the data has been at least buffered on the receiving host. This type of blocking communication, with receive buffering, most accurately represents the PDES and queueing model. In PDES the input waiting rule states that “a LP does not process any input messages until it has received at least one message from each of its input channels”[7]. Using non-blocking receive calls might violate this principle, leading to causality errors. As the LPs all operate independently, once the message has at least been buffered by the receiver, regardless of whether a matching receive call has been made, the sending process can continue regardless.
however this data must have been at least buffered by the process.

### 4.1.2 Message Packing

In the queue simulation software events are represented by a struct that contains the underlying variables used for PDES along with a user defined buffer, filled with data, to represent the event data that is scheduled in queues. Sending this struct via message passing can be achieved in a number of ways. The most primitive solution would be to enact a separate send and receive pair for each member variable of the message. Whilst this would be simplest it would also increase the number of MPI messages required to a send a single virtual message five fold. This wouldn’t accurately portray the normal service in a simulation model and I need to be able to send one virtual event in a single MPI message. The MPI standard dictates a range of data types and preparation methods that can be used to coalesce data into the required format so that it can then be sent with a single message. The message object that is passed between LPs contains variables of different type and varying length. To send this data with a single MPI message I’ve used a special data type called MPI\_PACKED. This allows the program to manually pack data into a user controlled buffer, using a MPI\_Pack call for each variable that is to be included, which can then be sent as a single message. This packed data buffer is received, using a single MPI call, and unpacked into the desired data structure using separate unpack calls for each element. The MPI\_PACKED data type allows me to easily send data for various types and lengths using a single message. This is an appropriate simulation of the queueing, and the underlying PDES, model.

### 4.2 The software simulator

The physical representation of the queueing simulator is discussed through this next subsection of the report. A basic outline of the program flow is given along with any other important implementation decisions that were made.

#### 4.2.1 LogicalProcess

This class represents the LPs, that are used to build a queueing simulator, in the PDES model. Following on from the design this object represents a sole LP, as described in PDES, with no actual MPI code for sending and receiving messages present. The actual message passing details are kept separate, in a container class, and this class simply calls the relevant methods of its container object to move events
around. This way any changes that need to be made are transparent to this class, ensuring maximum flexibility.

The main body of code runs within a loop that starts by checking for messages on its input channels, therefore conforming to the input waiting rule, and any messages that are received being directed into the LPs future event list, which is an ordered FIFO queue structure. After this the LP processes all messages in its future event list that have a time stamp of less than, or equal to, the lowest time stamp of all the messages received. From here each message is processed and sent on to the next LP, through an output link, using a round robin approach for distribution through each output channel. After all messages have been processed the LP then sends out null messages to every LP for which it is connected to but hasn’t sent a message to so far. This is accomplished by calling the sendNullMessages method of its container object with one argument which is the local virtual time of this logical process plus the lookahead value added on. In this case the lookahead value will always be equal to one virtual time point, since messages in the system are generated at discrete intervals, one time point apart. Any message that will be received in the future will have to be at least one time point past the current local virtual time of the logical process, which is equal to the lowest time stamp on any message received, and so this knowledge is used to calculate the lookahead value beforehand. Null messages, to avoid deadlock, were implemented as a later feature of the software as a prototype milestone.

The first LP in the queue acts as a generator, creating messages to input into system at set intervals. In my experiment the LP with rank 0 is used to generate the messages as it has no input channels. Instead of checking its input channels at the start of the program flow it calls a generating method which creates a new messages, with a changing payload, and stores the result locally. This new message is then sent on to the next queue in line using the standard send method, called on the container class. This process continues, generating messages at set intervals, until the maximum number of messages to be generated, at specified intervals, is reached. At this point the LP shuts down after it has sent this last message. As this last message propagates through the queueing system, each LP will increment their message count and upon checking against the maximum message count, shut down after passing it on. The final LP with no output links simply outputs the message’s content and finishes.

The event arrival rate, which was decided in the design phase, is set at 200 events a second. The actual delay is implemented by calling the **usleep()** function which halts a programs progress for a number of microseconds given as the argument. The actual argument is generated according to a Poisson distribution curve, so that the mean argument will be 5000 microseconds (which equates to 200 events
every second). Using a Poisson based arrival of events into the system increases the realism, events in a real system would be unlikely to arrival uniformly every 5 milliseconds.

4.2.2 MPIContainer

This class acts as a container which has a single LP member variable and takes care of all the MPI commands involved in the process of passing and receiving messages. This class has a set of methods such as send, receive, sendNullMessages, which act as the public methods for each LP to call when performing the actions involved with PDES. When the object starts on each host it initialises a private LP member, from which it then calls a runLoop method, which contains the main processing loop that runs the simulation exercise. This class decides where the messages get passed, through the use of message vectors, and also keeps track of messages sent for null message propagation. At the program start up a file is read in and processed which contains the link information that allows messages to travel between LPs. This file is parsed and each line is processed by the relevant LPs, dynamically creating the output and input channels that messages can travel through. When a message needs to be sent the class distributes the message along its input channel using a round robin approach. Each time a message is sent down an output link the actual link number is removed from the null message vector, any remaining links have null messages sent to them when this method is called. A null message is just a message with no data and is used to prevent deadlocks through the use of lookahead. Receiving messages is done in the same way, messages are received using the MPI recv() call and all messages are pushed in the LPs Future Events List, before the method returns. The timing calculations are measured in this object, on the process with rank 0. Two measurements are taken, once before the runLoop method is called and then after it has returned. Coupled with the use of barrier calls for synchronisation it then calculates the time taken for simulation, outputting the value to the command line.

4.2.3 Message

This class represents the events in my queueing model that are passed between the individual LPs in my system. The Message class has a Type member, indicating whether this message is an actual event containing data or just a null message, a TimeStamp member, which is set to represent the local virtual time of the LP it is being sent from, a DataLen variable, which is used to specify the length of the data in the message and is used when receiving the message and finally a Data member that is a pointer to the message contents. The actual message data is dependent on the data size that’s been specified for
the individual experiment. The initial data payload of unsigned characters is represented by creating a
buffer of the size specified by the data_size constant and filling it with the character ‘a’. Each time a
message is generated, at process 0, the character string is incremented once so that the data in the new
message will be different to the last one e.g. aaaa will become aaab, abcd will become abce and so on.
When a message reaches the final queue, having no output channels, the message content is outputted
along with the message count so far. This allows checking of the system for correctness in that it is
obvious if the message contents has been corrupted. Each message contents should be equal to the last
one when incremented once and the initial message will be a string of ‘a’ characters repeated.

4.3 MPICH

Compiling and running MPI programs, using the MPICH implementation, required a series of steps and
associated procedures to prepare the test bed for the experiments. The MPI simulator was first compiled
on each host using the mpiCC program, which is a complex script that uses a standard C++ compiler on
the host machines along with a series of MPI libraries to generate the MPI ready executables. Another
command mpirun is used for executing the processes on each host. This command takes a number of
arguments which dictate how many processes should be started and on which machines they shall run.
The number of processes is given by a standard command line arguments whereas the distribution of
the processes over hosts is slightly more complicated and resides in a separate file, called the machine
file, which is then given as a command line argument. This machines file contains the individual host
names for computers that processes are to be run on and in my simulation I also used this file to label the
distribution of processes on each host. The system has nine overall processes, five on iri01.leeds.ac.uk
and four on iri03.leeds.ac.uk. The command to then start the simulation is (where object_state is the
name of the simulator):

mpirun -machinefile machines -np 9 object_state

Each host which the simulation exercise is to be run on must have a compiled version of the simulator,
in the same directory and program name, for which it is initially invoked. The startup of each process
on the host computers is invoked using the Secure Shell Server (SSH), which allows remote login and
provides a shell through which commands can be executed. To automate this process SSH can use
a public key pair to enable automatic password-less logins between hosts. Once a key pair has been
generated, without a password, the keys are then copied into a special folder on each host. When the
mpirun command tries to log into each host it can then use these keys to authentication rather than having to request a password.

Although mpirun will distribute the individual processes over the hosts as I’ve requested, the actual mapping of process IDs to hosts is dependent on the MPI implementation. For the experiment to be most effective each MPI message should be sent over the network rather than through internal channels and subsequently the queue links between the LPs need to reflect the actual topology. Using a sample program I created, that when ran each process printed out its rank ID and the host it was currently running on, I was able to work out the distribution of process IDs between the two hosts. The simulator topology is read in from a text file which contains the link information between LPs in the system. Once the actual process distribution had been calculated it was a simple task to create a link file which mapped to physical topology to a virtual one.

4.4 MPICH-G2

MPICH-G2, a Grid enabled version of the MPICH library, has the benefit of allowing MPI code compilation and execution in exactly the same way as normal MPI code is run. To compile and run MPI programs on a Grid all that is required is that the MPICH-G2 versions of mpirun and mpiCC are used, in the same manner as the MPICH tools. A slight change in the command line arguments given when running the MPI programs, via mpirun, is the only slight difference between the two procedures. The MPICH-G2 implementation of this command uses the Resource Specification Language (RSL) to describe the job. Users write RSL scripts, which identify resources, requirements and parameters for the job. Based on the information found in an RSL script, MPICH-G2 calls a co-allocation library distributed with the Globus Toolkit to schedule and start the application across the various computers specified by the user[32]. RSL scripts can be used to directly dictate the mapping of process IDs to hosts, so the queue links file can be created in advance without any direct experimentation needed to discover this information.

4.5 Globus

Running any Grid enabled MPI applications requires the user to be able to provide some form of authentication for the unique name they are signing on with, granting them access to resources on the Grid. All users are identified by a globally unique name known as a “Distinguished Name”. Each entity has two
keys, which are the public key and the private key. The public key is encapsulated in a X.509 certificate that includes the distinguished name, public key and other things, and is signed by a trusted party.[38]. Generating a key pair, which is then used to form part of the certificate, is completed by running the following command and setting a password: grid-cert-request -ca dcdd319e. Once a certificate request has been generated it then needs to be signed by the certificate authority, here given by the command arguments. After getting the user certificate signed it can then be used to authenticate the distinguished name of the certificate with the local Grid, using the given password chosen before when creating the key pair. To allow applications to use Grid resources for a user, authenticated with their credentials, Globus allows the creation of a proxy, “A proxy is a short-term credential that is created by a user to authenticate that user.”[38]. Applications can then use this proxy to enable them to access resources for a limited duration of time. To create a proxy the following command is issued, grid-proxy-init. A variety of options can be set on initialisation, such as the proxy’s time span, and once the user’s password is entered the proxy is created. The current status of the proxy can be checked with grid-proxy-init and it can later be revoked with grid-proxy-destroy.

4.6 Software Development

4.6.1 Methodology and Process

The software simulator was developed using the iterative prototyping model, through the use of rapid prototyping combined with target milestones, which allowed an iterative approach to software development in the project. The project started off with specific milestones that related to the minimum aims which were laid down at the start of the project, that were:

- Implement a simple discrete event simulator to simulate a queueing network
- Modify the simple discrete event simulator to incorporate one parallel discrete event simulation protocol
- Extend the parallel discrete event simulator to run on a Grid

Each step represented another stage in the iterative development process. Development couldn’t proceed until the whole process had completed on the previous stage. Developing software in this fashion allowed me to introduce testing of the simulator early, helping to catch any initial errors straight
away. I also built up the inclusion of specific technologies using a gradual process so that I became confident using MPI, Globus etc. from an early stage in the project. In the beginning the different versions of the MPICH suite and the Globus toolkit caused some teething problems, due to incorrect configuration, which manifested themselves in problems when I tried to run any software. Following correspondence with the local support administrator it became apparent where these problems lay and they were promptly fixed. Using an object-orientated approach allowed me to build and test the classes involved separately, to ensure correctness, before I combined the classes together to form a complete application. This iterative approach was a great advantage when learning a plethora of new technologies and libraries, helping me build up my knowledge in incremental steps.

As part of the prototyping model, once prototypes were developed they were then evaluated and the feedback from this stage was then integrated back into the initial system’s design, refining the process to create a more appropriate design based upon the results of testing. My initial system design as shown in figure 6.8, in Appendix D, evolved through this process, eventually finishing up as the final design shown in figure 6.9, in Appendix D.

### 4.6.2 Platform, tools and environment

The parallel discrete event simulator was going to utilise the MPICH implementation of the message passing interface due to the familiarity of the library within the School of Computing. This software library was already available and configured for use on the laboratory computers, that were to be used in the testing. MPICH provides libraries and tools for a number of languages including C, C++ and Fortran. Having no previous experience with Fortran meant that the C or C++ was going to be the language of choice for the software. Object-orientated design and development is the methodology which I have followed throughout my time at University and was the obvious choice when deciding how I should design the software application I was going to use for testing.

C++ was designed as a language which follows the object-orientated approach to programming. Whilst C can be used when trying to program software in an object-orientated way, the language isn’t naturally geared towards development of this type. My familiarity with C++, which I’ve used extensively at University, and its naturally object-orientated approach to development made it the most suitable language for the software development aspect in this project. The experiment’s computers, which were to be used, both ran Unix derived operating systems (Linux and IRIX). This meant that the software development was easily accomplished via remote login using SSH to access login shells on each ma-
chine without needed physical access. The main glut of development took place on iri03 as this was a Linux machine, an operating system I was more comfortable with. The ease with which MPI programs are run and configured meant that rigorous testing took place often, which every new feature, using solely the local machine and then both machines, normal MPI and MPI over Globus. My familiarity with the Linux operating system meant I was able to use a whole suite of standards tools when developing the software simulator. The Vi Improved editor, Vim, with a standard bash shell and mpiCC, the MPICH wrapper around the host’s C++ compiler, comprised the main development platform that was used. Other standard Unix tools such as grep, cat and sed were used occasionally when required.

During the project I created a series of small bash scripts to help out with maintenance procedures, such as backups. Setting up an individual experiment required a number of independent steps, such as changing the message count in the source code, compiling the source on both hosts, starting the traffic sniffer and so on. Each step had to be performed in a designated order, failing to complete a single step would cause the experiment to fail at some point. Some of the longer experiments took nearly an hour making it imperative that everything was setup correctly beforehand. By creating a series of simple scripts in the bash language I automated a large part of the configuration for each experiment which meant that each test ran correctly and could largely be left alone whilst it ran. Once the test had completed I simply noted down the results and started on the next test. Other important aspects of the project, such as the backing up of results and source code, was automated using another bash script with the tar compression tool to archive everything I had done at set intervals. After the data had been collected it was then processed using a collection of small Python scripts to prepare it into a suitable format. This prepared data set was then import into Couplet, which is an open source tool for generating graphs. A collection of different graphs were created, using GnuPlot, to help in the evaluation.

4.7 Capturing experimental data

Using time as the obvious metric for measuring the performance of MPI then leaves me with other questions. Where and how in the experiment should I measure time? I must take into account a number of extenuating factors that will affect the overall time taken for an MPI program to execute. Startup and shutdown time may vary widely, dependent on various external influences, yet this delays happen when the programs aren’t actually sending any data and obviously need to be accounted for in the overall results. The time needed to distribute and run the various processes may be far higher on the Grid due to
initial start up costs however most parallel programs will run for a long time, making these initial costs
irrelevant to the overall performance measurement of the system. The incurred times might also vary
widely between MPI and Grid tests, for reasons such as machine load or resource allocation on the Grid,
and so they can’t be incorporated into the final results as a linear constant on top of the actual message
passing time.

4.7.1 Inappropriate measurement methods

Using a simple timer on the controlling host, such as the Unix command \texttt{time}, at the operating system
layer wouldn’t be able to automatically account for the intricacies of MPI. Using this approach, seeking
to time the entire program execution time at a lower level than the MPI constructs will lead to inaccurate
results or need manual adjustment of the data, using approximate methods in calculating the associate
MPI startup and shutdown costs. Attempting to measure program time at an even lower level, the
network layer, by reading the packet data leaving the controlling host would bring its own problems. It
would be hard to use a simple packet sniffer as the sniffer would have no knowledge of the MPI protocol
and couldn’t differentiate between the initial transfer of programs and the actual message passing. The
packet sniffer would have to be started at some point after the MPI setup. Packet sniffing could however
be used to measure the performance, of MPI, in terms of data transfer and bandwidth utilisation of the
underlying network. This could later be used along with the timed results to produce a better overall
picture when measuring performance. Directly reading data off the wire will give a better indication
of the data sizes being sent over the network. Calculating this without doing any direct measurement
would require knowing the exact packet layouts for MPI, not just the messages sizes that are being
sent. Extending this to incorporate the Grid network layers would be far more complex and becomes
unrealistic. Measuring data flow straight off the network interface will give us direct access to packets
as they would be sent across the network, already encapsulated in the specific MPI or Grid layers.

4.7.2 Measurement solution

The best possible solution will be to implement the timing functionality as part of the queueing system
itself. Situating the timing methods within the MPI program allows me to overcome the problems
described above. I can easily tune the timing procedure to just measure the core activity of the program,
ignoring everything else that’s deemed irrelevant. This approach is also extremely flexible. The MPI
library provides the functionality needed in recording timed measurements at specific points in the
programs flow. The function MPI_Wtime return a long integer specifying the number of seconds that have passed since a fixed time point in the past. The specified value doesn’t change and therefore this allows me to call the function twice, storing the results, before and after the code that’s being timed. Subtracting the first value from the second will give us, in milliseconds, the execution time of the procedures in between. A similar function, MPI_Wticks, will perform the same function but returning the number of CPU ticks since the fixed point in time, for higher granularity calculations. Using the MPI_Wtime function will allow for time measurements down to the millisecond which is accurate enough granularity for the experiments.

Another MPI function, MPI_Barrier, can be used to enforce temporary synchronisation between the individual processes. “A barrier is a special collective operation that does not let the process continue until all processes in the communicator have reached the barrier call in the program execution”[16]. In the simulator I have two barrier calls, at the beginning just before the first message is generated and then again at the end after the final message has been passed into the system. The two calls to MPI_Wtime occur before the first barrier call and after the second, thus ensuring that the time points recorded encapsulate solely the simulation exercise and nothing else. I can use this method to ignore any initialisation or finalisation overheads that may occur and aren’t relevant to the experiments.

Aside from time I’m also measuring the data transferred between hosts during the simulation exercises. The experimental test bed only uses two computers and this fact makes recording the data transfer between in the simulation extremely easy. A traffic sniffer is used on one of the hosts and captures all the traffic travelling over a network interface, writing the captured packets to a file. This file can later be analysed as an exact copy of all network traffic that occurred in the simulation. TCPDump is a well known traffic sniffer for the Linux operating system. Using this software I can accurately record only certain parts of all the network traffic that travels out of a network interface on the host computer. Before the simulation was started I started TCPDump as the following command line:

```
tcpdump -i eth0 -w capture.trace -n -s 0 “host iri01.leeds.ac.uk and (udp or tcp) and ( (tcp[2:2] > 1024 and tcp[0:2] > 1024) or (udp[2:2] > 1024 and udp[0:2] > 1024) )”
```

This captures all UDP or TCP traffic, between hosts iri01 and iri03 on the primary network interface, that uses a non-privileged source and destination port (over 1024) and writes the resulting data to a file in binary format. This capture would include all of the MPI messages in the simulation but could include some over network data unrelated to the simulation, however this affect should be minimised by using such an explicit traffic matching expression. The fairly large amount of data transfer involved with all
my simulations means that a small number of external packets would be extremely unlikely to have a noticeable impact on the results.

4.8 Problems

During my development phase I was using the iterative prototyping process that meant that I was constantly prototyping new versions of the simulator, following an incremental series of milestones, testing these new versions and correcting any flaws. When the simulator got close to completion I began to run some initial test experiments in the conditions that mirrored the real simulation exercise configuration. Whilst everything worked fine for small message sizes when I began to increase the message size variable, coupled with a large arrival rate, the system became unstable and produced erroneous errors.

After more testing it became apparent that somewhere the software was having problems coping with the extremely fast heap memory allocation and deallocation, approximately two hundred times a second, and this was causing intermittent failures in some of the processes. Once one process failed, due to the topology of the experiment, the whole system came to an abrupt halt as a link in the queue stop functioning. This point came at a late stage in the software development, when I expected any serious errors to have already been caught in earlier versions. Subsequently, after further investigation, I decided that I'd have to redesign my experiment to work around this problem because of the time shortage.

My initial investigations into the problem produced a threshold value for the message size that could be used without causing the system to run out of memory, which started to appear after the heap allocated message data grew above 20 bytes, making the total message object nearly 50 bytes in size. The message object’s size was then set to 28 bytes, by reducing the message contents to 10 bytes, and with this message size the system worked without any further problems. Although these values seems relatively small it became a problem when competing processes are trying to allocate, access and deallocate dynamic memory in such a rapid fashion. The errors that were produced proved extremely unintuitive and changed between runs which made debugging the problem extremely problematic.

Due to the second variance in my experiment, the message size, being fixed I wouldn’t be running as many experiments and to compensate for this choose to increase the range of total messages that I was testing with. Originally my experiment was to use total message counts of 5000 up to 50,000, in graduated steps of 5000. I felt that my experiment could now benefit from a extra set of larger
message totals that could be used to test the system for extended periods of time. Individual steps of 100,000, 250,000 and 500,000 were now included in the experiment values to be tested for. Using larger graduations allowed the experiment to test whether any initial patterns that are observed carry on at larger intervals. These larger values will also run for a considerable amount of time giving the experiment a thorough test.
Chapter 5

Results

This chapter contains the results for the experiments that were ran and analyses the data recorded, in order to draw conclusions about the performance of parallel discrete event simulation on a Grid test bed.

5.1 Raw Data

See Appendix B for results tables and all associated graphs.

5.2 Performance analysis

5.2.1 Main metric

Judging the performance of the simulation experiments solely against the main metric of time, that was taken to complete the simulation exercise, one overriding trend emerged. It became evident as the experiments were being ran and whilst the data was collected that the performance impact of running the experimental PDES software, running as a queueing simulation on Grid based MPI and compared with traditional MPI, was negligible. In all the tests that were ran, direct comparison of the results for each run, demonstrated that the difference between the highest and lowest simulation time from either test condition (MPICH or MPICH-G2) was never more than 2% apart. In the majority of cases the difference between simulation runs, for a given message total, was in the order of a few hundred milliseconds and this trend carried on even as the amount of messages passing through the system became extremely large.

As the total messages count, that was being varied in the experiments, was increased the simulation
time increased in direct proportion to the overall message count. The initial experiments using 5000 messages, using both platforms, produced a mean simulation time of approximately 32 seconds. The arrival rate at process rank 0 was set to be 0.005 (seconds per message) and this meant an average of 200 events were being generated every second, following the Poisson distribution curve for event generation. A pure, virtual system with no overheads associated with actual message passing would take 25 seconds to simulate 5000 messages passing through the system (assuming the time for an event to pass through all queues and exit the system was negligible). Both Grid and normal MPI based simulation exercises for the initial 5000 messages incurred approximately seven seconds worth of overhead when compared with the ‘perfect’ system.

As the experiment progressed in steps of 5000 it was easy to observe how the system scaled with larger values following on from the initial experiment. In Figure 5.1 it is clear that both experiments for each subsequent value follow a linear scaling for larger message totals, that 5000 messages take around 32 seconds of simulation time regardless of which system is being used. The results for both platforms at a large value of 100,000 messages returns the mean simulation times of just over 651 seconds for both MPICH and MPICH-G2. This is an average of 32.55 seconds required for simulation of 5000 messages passing through the system. Grid based MPI follows the pattern established by traditional MPI, that simulation time increases along a scale that directly follows the proportional increase of the total amount messages being simulated. This pattern is evident throughout the test results and carries on as the message totals increased a hundred fold to 500,000 for the final experiments.

As the total messages variable increased to relatively large values the actual deviations between the
recorded simulation times, for normal MPI and Grid based, grew proportionally to the overall simulation time, which is evident in figure 5.2. For example the third run for Grid based simulation over 500,000 messages returned a simulation time of 3285.33 seconds, whilst the other two results for this value produced times of 3246.69 and 3241.03. Whilst this value may seem like a large variance in comparison to the MPI mean time of 3240.674, a total divergence of 44.656 seconds, as a percentage it only represents a change of 1.38%, which is in line with the other ranges of results seen in smaller tests. The third run for 25,000 messages using MPICH-G2 yields a simulation time of 163.72 which is in fact a 1.41% percent increase on the MPICH mean value, larger than the variance displayed for the apparently large difference recorded at 500,000 messages.

In the majority of cases for the MPI simulation results the actual variance between runs was extremely small, in the order of a hundred milliseconds and pattern continued even as the total simulation time increased. Looking at the standard deviation for the MPI results it’s clear that it doesn’t follow such a direct scaling in proportion with the overall simulation time. The results for the Grid experiments show that standard deviation between results was relatively small and that the range of results returned was again extremely similar. However as the message totals increase, whilst the standard deviation of the MPI tests remains generally under hundred milliseconds regardless of the total simulation time, the Grid based results do demonstrate a more proportional standard deviation to the overall simulation time. The standard deviation for Grid results reaches above a second in three cases (25,000, 30,000 and 50,000) for the tests from 5,000 up to 50,000 whilst the standard deviation for the MPI results remains under a hundred milliseconds for these values. When the tests involved the larger values of 100,000 and
more, standard deviations of even greater values are present, although when put into context they are still extremely small in comparison to the overall simulation time.

The range of results for each MPICH run in the experiments were generally extremely small and one effect of using the Grid platform for MPI was this larger range seen, especially at the extremely larger message totals that were ran. Although the results show that the performance impact overall was fairly negligible, when juxtaposed against the overall simulation time, another pattern that was evident is that the mean simulation time for each platform indicated that Grid based MPI was marginally slower. In all cases the mean time for Grid based MPI simulation was always higher than the normal MPI platform time. However the difference between both times was generally extremely small, as previously stated, and represented less than a 1.5% change between the simulation times, making the overall performance impact of MPICH-G2 on the simulation experiments minimal. Another important factor is that although the results demonstrate a possible performance impact of under 1.5% this performance impact is not a certainty. Looking at the results for a 100,000, a large value where any performance impacts would have time to emerge, the actual difference between the mean times for each experiment was 0.15133 seconds, a total change of 0.023% for MPICH-G2 against MPI. The range of results for the MPICH-G2 runs was also minutely small, providing evidence that the supposed performance impact is a possibility but not necessary always evident.

Figure 5.3: Mean data transfers for 5,000 to 50,000 messages, using the MPI and Grid platforms
5.2.2 Secondary metrics

The evaluation of Grid based parallel simulation, using the primary metric of simulation time as the performance indicator, has shown the impact of a Grid environment to be minimal in the experiments. As well as collecting data to record the main benchmark, the simulation time, the experiments also collated a large amount of evidence that can be used to evaluate another aspect of the system performance, the data transfer required for the simulations. Using the tcpdump application all the packets sent between the relevant hosts involved in each experiment was recorded in a format that could later be analysed. These “packet dumps” were later ran through a series of other open source programs to extract information from the raw data. Using these programs the number of packets and the combined size of all the packets in the trace file were derived and then recorded.

Taking the data transfer between hosts in each simulation experiment as another metric on which to judge the performance of Grid based parallel simulation provides a vastly different conclusion to my previous one. It is obvious from looking at the data and visualisations produced that the previous overriding trend observed, Grid based simulation has no major impact on the simulation time, isn’t also evident here. For both sets of data, the total data transfer for each experiment and less importantly the number of packets involved, Grid based simulation produces results which are far larger than the traditional MPI simulations. In figure 5.3, between the steps of 5,000 to 50,000, the relationship between the Grid and normal MPI simulations isn’t a fixed addition on top of the normal MPI results but in fact a proportional linear scaling. Looking at the raw data for the Grid simulation, data transfer is on average approximately 2.3 times as large as the MPI results. This value stays constant throughout the graduated steps and is evident in figure 5.3.

Analysing the data for the larger values shows that this apparent trend continues in the simulation results for 100,000, 250,000 and 500,000 messages. In figure 6.5 this pattern is prevalent right throughout the experiment’s results and isn’t particularly influenced by the total simulation time. Due to the large range of figures involved figure 6.5 has logarithmic axises. The range and standard deviation of results for the MPICH simulation runs, for a particularly message total, were extremely small and were all within less than a single percentile of the mean. The MPICH-G2 simulations also followed this trend and the runs for each experiment returned values that were usually within a percentile of the mean. Furthermore, as has been demonstrated when looking at the data for the simulations times, Grid based simulations did tend to produce simulation totals with a slightly greater standard deviations of runs than the requisite MPICH runs, even when the standard deviation was scaled to take account of the increased
The other piece of data that was extracted from the trace files for analysis was the number of packets recorded for each experiment. The mean number of packets involved in a simulation of 5,000 events was 71,487 and this number followed a proportional scaling as the number of events grew, 50,000 events have a mean packet total of 708,455 a ten fold increase. The Grid simulation results were far larger than the MPI totals and not surprisingly followed a relationship similar to the one observed for the overall data transfer. The Grid simulation’s mean number of total packets, for each particular value, followed the proportional scaling effect of over double the MPI mean result, that corresponds to the findings for the total data transfer involved documented above. This scaling effect is also seemingly unaffected by the overall simulation time as graph figure 5.4 demonstrates the pattern continuing with the same factor for each step. One slight difference between the total data transfer and total packet count scalings, in proportion to the MPI mean results, is that the total packet count is slightly lower. The mean total packet count is on average approximately 2.1 times the size of the MPI total as compared a factor of 2.3. Unfortunately due to the extremely large size of the packet traces for the steps of 100,000, 250,000 and 500,000, in the order of gigabytes, I was only able to record the total data transfer size before having to remove the files for space concerns on the host environment and as a result haven’t been able to run the files through further analysis tools to extract the total number of packets involved. Looking at the results for the total data transfer and the steps of 5,000 to 50,000 for the total packet counts I believe that this trend would no doubt continue at the same rate for those larger values, figure 6.5 demonstrates that the total data size continues along with the trend for the smaller steps.
Chapter 6

Evaluation

This section will analyse a number of different aspects of the project in an attempt to evaluate the performance of parallel discrete event simulation on the Grid. Principally the main aim will be to evaluate the test data in order to draw some conclusions about the performance of my application over both platforms (MPI and MPICH-G2). Further more a wider ranging evaluation of the whole environment and project aims, not just concentrating on performance, will help to understand whether parallel simulation on the Grid is also feasible in terms of the actual running and building of parallel simulation applications. Finally a personal evaluation on the project from my perspective, that will hopefully allow me to demonstrate any further information and suggest improvements.

6.1 Performance of PDES on the Grid

This project’s main aim was to investigate whether parallel simulation using the emerging Grid technologies available was a viable and feasible proposition. To accomplish this I built an example a sample PDES application that was used to simulate a queueing network, a traditional method for modelling in parallel simulation exercises. Utilising available resources I was able to run the software using normal MPI methods and MPI over Globus. Following on from this I designed an experiment that would accurately represent a real world simulation exercise and used a variable quantity that would be changed for each different test, data relating to the performance of the simulation under both conditions could then be collected and evaluated. Time was chosen as the main metric, by which performance was going to be judged, due to the ability of it in reflecting the performance of a whole range of issues in the system and also being able to exactly quantify what was to be measured in the experiment.
The experimental stage performed a series of rigorous and thorough tests which ran the software simulation exercise for a large range of different times and total messages. Each experiment was performed on three individual occasions and in most instances the tests for the same value weren’t ran in a back to back sequence in time. This reduced the impact of any uncontrollable environmental factors on the results, as all the experiments for one value were unlikely to be in affected in the same manner. Initially the tests ran for a series of graduations going from 5,000 messages up to 50,000 at steps of 50,000. Another series of three additional steps was then added of 100,000, 250,000 and 500,000 to give the system an extremely long and extensive test, allowing an analysis later on to see if any initial trends carried on for extremely larger values.

6.1.1 Performance evaluation against the main metric

In section 5.2.1 the data obtained from the experiments was analysed against the main metric, the overall simulation time required, and as stated one overriding trend emerged. Against the primary metric, that was set out when first designing the experiment, the performance impact of using a Grid test bed for parallel simulations was minimal. The results show that the variations between the results for both platforms was, in the majority of cases, not more 1.5% and this remained prevalent even at the larger message counts. This is an extremely small value, considering the extra complexity that Grid computing brings, and can be judged to be a considerable success.

6.1.2 Performance evaluation against secondary metrics

The secondary metrics of data transfer and number of network packets, as analysed in section 5.2.2, showed a different set of conclusions however. The performance impact of using a Grid based platform show an increase of over 100% when compared to traditional MPI in the amount of data transferred and the number of packets used. This relationship between both platforms did stay static as the message count used in the experiments increased, the factor increases for data transfer required was an average of 2.3 and the packet count of 2.1 against the results for MPI.

6.1.3 Evaluation of results against related research

The rapid emergence of Grid computing has meant that the technology behind it is still reaching maturity and as a result, research into the use of Grid computing as a platform for traditional parallel simulation is still a fairly new area. Whilst the amount of research into this area isn’t substantial as yet there have
been some initial investigations carried out to assess this new and exciting opportunity. In Alfieri[39] a performance evaluation of MPICH-G2 on a cluster of PCs was carried out. The experiments evaluated the performance of MPICH-G2 versus MPICH using a collection of standard MPI performance tools included with the MPICH library, mpptest and goptest, used on a four node cluster connected over local and wide area networks.

The tests carried out a series of operations from simple ping-pong tests, testing point-to-point communication, to reduction operations, testing global collective communication, under a range of different conditions. “LAN tests were performed between two different machines on the same Fast Ethernet LAN without shared memory support.”[39], which closely reflects the environment my experiments were ran in. “The results show a higher latency of MPICH-G2 with respect to MPICH”. Latency is an extremely important factor in parallel simulation, especially using such a fine-grained simulation exercise as I was conducting, and this would correlate with the slight increase in simulation time that was observed when using MPICH-G2 in the experiments, due to increase in data being sent. Due to earlier problems in the implementation of my experiment I had to reduce the message sizes being sent to a relatively small amount, in Alfieri[39] “the relative figures shows a better performance of MPICH with short messages (100 byte)”. The discrete events in my model were simulated using small messages, less than 100 bytes, and the results also showed that MPICH performed marginally better than MPICH-G2. It is apparent from reviewing the figures and tables in this paper that, as with my experiments, the performance decrease from using MPICH-G2 is relatively small for many of the tests on a LAN.

A more thorough and comprehensive evaluation of MPI implementations on Grid clusters is carried out in Matsuda [40]. In this paper a variety of MPI implementations, for use with cluster computing, are evaluated against each other. In their experiments a wide area network is used for MPICH-G2, consisting of 32 hosts alongside a variable network environment. The conditions such as greater latency are simulated using a specialised Linux kernel module. The bandwidth tests used a large number of different messages sizes, however the most relevant for performance evaluation with my tests is messages below 16KBytes, messages above this threshold value are delivered between hosts using a different protocol, called rendezvous, instead of the eager protocol which was used in my experiments due to the messages being below 16KBytes. “Comparing the performance of MPICH-G2 with MPICH reveals the overhead for security and adaptation to a Grid environment. Although the performance is slightly disturbed in MPICH, mostly it performs better than MPICH-G2”[40].

The latency experiments performed later on reveal that MPICH-G2 doesn’t perform as well MPICH
although the manner of the performance degradation is dependent on the type of parallel simulation test being ran, in the case of an embarrassingly parallel problem, no apparent performance impact is observed. However in later tests, where the parallelism is far more finely-grained such as Figure 10 in the paper[40], it is obvious to see that MPICH performs much better in such a scenario. The performance impact of MPICH-G2 varies according to the experimental setup and in many occasions the difference between MPICH and MPICH-G2, whilst evident, isn’t of major proportions. This conclusion is something that can be observed in the experiments I performed when judging performance against the main metric of the overall simulation time, whilst the a performance impact did occur it was still extremely small in comparison to the whole system.

6.1.4 Sources of error and problems encountered

The original experiment was rigorously designed to fully test parallel discrete event simulation on the Grid, taking into account sources of error and possible pitfalls that could occur. A whole series of possible problems associated with the performance testing of MPI implementations were listed in Gropp and Lusk[35] and were mitigated as far as possible. Unfortunately during the implementation stage, when the software neared completion, a previous unforeseen error and extremely serious bug began to appear. The problems arose when the message objects grew above a particularly size threshold, as this point the system became unstable and the individual processes were likely to crash bringing the whole experiment to an abrupt halt. Varying the message size was an intrinsic part of the original experiments and designed to test the system for a range of different conditions, more importantly when the MPI messages being sent grew larger than 16KBytes the underlying protocol used to send the data changed from eager to rendezvous. The eager protocol simply sends the data to the destination regardless of whether a matching receive call has been posted, whereas “in the rendezvous protocol, data is sent to the destination only when requested (the control information describing the message is always sent). When a receive is posted that matches the message, the destination sends the source a request for the data.”[37].

Due to the late stage in the project lifetime at which this error occurred and my relative inexperience with parallel programming, despite help from other people and a lot of time spent debugging independently, I wasn’t able to produce a reliable fix for the program. Instead I decided to redesign the experiment to cope with this limiting factor, choosing to simulate a much larger range of total messages with a fixed size message. Unfortunately, due to this problem, it did narrow the range of systems that
I could simulate and therefore draw conclusions for from the data I collected. All the experiments I performed were passing messages between the individual processes using only the eager protocol and in the real world this wouldn’t always be the true. Using an increased samples of message totals did however allow me to test any conclusions I reached for the initial range of 5,000 to 50,000 messages in much longer simulation exercises, meaning that any conclusions that I reached would have a large weight of evidence behind them.

Secondly the queueing model for simulation I used, that is widely prevalent in the real world, coupled with the fast arrival rate meant the simulation software had a highly fine-grained approach to parallelism. Parallel simulations of this type would have been unlikely to be sending messages, at such a high frequency, that were extremely large in size. More coarsely-grained parallel exercises would have been most likely to be using messages of large sized, at far slower arrival rates, and as a result I believe that the system I created would in fact represent a large number of real world systems that I was supposed to be partially testing for.

6.1.5 Conclusions about the performance impact of PDES on the Grid

This project has been run with the purpose of trying to judge the performance of PDES on a Grid test bed. A series of experiments were designed and ran, using a custom built parallel discrete event simulator, to collect data that could then be used to judge the results against a set of predetermined criteria. The main aim of this project can be summed up by a single question “Can parallel discrete event simulation utilise the emerging Grid computing platform successfully?”. Whilst the previous research into the area approached the area from a different angle, the actual performance of the underlying technologies, I tried to simulate a scenario which would model the real world. Building a normal parallel discrete event simulator using normal parallel programming technologies and comparing the performance on both the respective platforms I hoped to answer the question of whether existing and new systems could easily be adapted to run on a Grid.

Using a Grid platform brought up a range of possible problems with relation to performance, the main one being that in parallel simulation “performance is dominated by latency”[41]. Grid computing is inherently distributed and this “implies significant network latency”[41]. The main criteria that was used to judge the performance of the system was the overall simulation of the experiments on both platforms. In my simulation exercises the performance impact, of using a Grid platform, on this criterion was shown to be minimal. Each exercise generally ran within a single percentage of the normal MPI
platform time observed. This trend remained true as the number of events in the system increased, even when the steps became extremely large with the last simulation experiment with 500,000 events taking nearly an hour. Judged against this main metric I can see that the Grid computing platform was of no serious hindrance to that specific performance measure. Other research into the related areas[40][39] has shown that under similar conditions the same results have been observed, MPICH-G2 does have an impact on performance but under certain conditions this impact is marginal.

As well as looking at the overall simulation time, as a main metric on which performance was to be evaluated, the experiments also collected other sample data such as the network data transferred in the experiments. Using other criteria based upon secondary metrics, such as number of packets used and amount of bandwidth used, does lead to a different set of conclusions. The data collected shows that the Grid platform produced over double the number of packets and amount of data transfer for each experiment when compared to the normal MPI platform. The overheads associated with Grid computing become apparent when this metric is used to judge the performance against. An increase of this scale is significant and could be a very important consideration when evaluating the suitability of Grid computing for PDES.

Despite this large increase in certain computing resources required, for the PDES experiments on the Grid platform, it had little impact on the overall simulation time. This is no doubt due to the low latency, high bandwidth test environment that the experiments were performed in. Although the data transfer more than doubled when using MPICH-G2 this increase was easily coped with by the fast Ethernet connection between the two hosts involved in the simulation exercises. The actual bandwidth required, even with the increase, was still less than the test bed provided and coupled with extremely low latency it meant that the increase in the number of packets being used wasn’t a problem.

In the experiments this possible performance impact was nullified by the test environments capabilities however under a set of different scenarios this could be a big problem. In Matsuda[40] the affect that large network latency has on the performance MPICH-G2 in comparison to MPICH becomes evident. In Grid computing distributed and heterogeneous resources make up a computing platform which can be used for parallel simulations. Unlike the test conditions my experiments were performed in, the possible network conditions of a real world Grid aren’t guaranteed to be so forgiving and “distributed simulations have specific characteristics: stringent end-to-end delay constraints and sensitivity to network delay jitter”[41].

In spite of the apparent possible problems that have been discovered during the tests I believe that
the main performance evaluation has shown that Grid computing is an extremely promising platform for parallel simulations to be run on. Against my main metric it has been shown that the performance is practically similar for the experiments and whilst this may not scale to larger scale Grids, running over WANs with high latency and lower bandwidth, for a large range of very appropriate conditions, Grid computing provides an excellent way to perform PDES. The benefits of Grid computing to PDES shouldn’t be limited to being judged solely on a single aspect of performance as “the emergence of Grid services and the potential for seamless aggregation, integration and interaction makes it possible to combine computations, experiments, observations and data to form a powerful simulation environment.”[41]

### 6.1.6 Further enhancements

If the experiments were ran again a number of possible enhancements could be made to enable the test to explore new performance metrics or see how different factors associated with MPI and PDES affected the results.

**Wide Area Network** The main extension that I originally planned on completing was the running on the experiments using the software on a Grid connected using a wide area network, like the White Rose Grid[33], instead of a local area network, which I used. A further enhancement to the experiment would be to actually run the experiment over a large scale Grid. Using a wide area network would simulate a large number of real world Grids, including the network conditions, that would no doubt have larger latencies and variable point-to-point bandwidths between the computing resources connected.

**Change MPI Protocol** After encountering some problems with the stability of the system when using messages of larger sizes, the experiment was redesigned to use a single message size that was relatively small, instead concentrating on the total messages. As previously written about, over 16KBbytes the MPICH library changes the protocol it uses when delivering messages from eager to rendezvous. Currently all the experiments I performed used the eager protocol, due to the small message size, and one possible extension would be to see how the system’s performance performed when using the other protocol, with the increased number of messages involved in sending data in the simulation.

**Number of LPs** In my tests a deliberate decision was taken to only use 9 LPs, so that each system in the experiment wouldn’t have more LPs than physical processors. This decision was made in order to reduce any external influences on the results, which could occur as a result of the context switching.
needed to run $n$ processes on $n - 1$ physical processors in the simulations. Further experiments could attempt to measure the affect this factor has on the system by increasing the number of LPs used in the simulation exercises.

**Arrival and service rates** A third possible extension to the experiment would be to vary the arrival rate of messages into the system. This would allow the experiment to test the scaling of the simulator, especially as the message rate increased and see if my initial findings were evident at far higher arrival rates.

### 6.2 Overall Project Evaluation

This part of the evaluation will take an overall view of the project and attempt to evaluate it as a whole, comparing the work completed and results against a set of criteria established at the start and encompassing a broad scope of different aspects in the project.

#### 6.2.1 Project aim

The aim of this project was “to evaluate the performance of parallel simulation on a Grid test bed”. My original background reading and research, followed by the design and implementation of a parallel discrete event simulator which was then used to carry out a series of experiments, the results of which were thoroughly evaluated, meant the ending of this project has signified the completion of the aforementioned aim. A series of objectives and minimum requirements were achieved during the project’s lifetime and evidence for these claims is demonstrated below.

#### 6.2.2 Project objectives

In the beginning of this project three individual objectives were set out, the completion of which would hopefully indicate a successful and worthwhile project. Taking each one of these objectives respectively I will demonstrate the work completed, therefore indicating the completion on that task.

- **Produce a software system implementing a parallel discrete event simulator using parallel programming in a Grid environment.** Using the prototyping development methodology I created a series of prototype simulators in software, each building upon the last until a version was reached that represented a fully featured parallel discrete event simulator. This final version used
the MPICH implementation of the MPI standard to utilise parallel programming technologies within the software. Once the software had been completed a Grid enabled version of the MPICH library was used to run the software on a Grid test bed.

- **Use this simulator to test performance of a parallel discrete event simulation on the Grid.**
  The software simulator was used for performance testing on a Grid test bed through the design of a series of rigorous experiments, designed to test the performance of a PDES on a Grid test bed against a series of metrics.

- **Evaluate parallel simulation on a Grid in comparison to traditional parallel programming environments.** Once the experiments had been completed, the data each test produced was collected together into a single format for further analysis. A variety of tools was then used to produce graphs, charts and extract more information from the raw data. These combined sources of information were then used to evaluate the performance of the experiments on both platforms and make a number of conclusions.

### 6.2.3 Minimum requirements and extensions

An initial set of minimum requirements were created at the start of this project which would form a series of milestones for completion, each showing that another stage of the overall project had been achieved and that progress had been made. Going over the initial requirements I can demonstrate that each of the milestones was completed and indicate how and where this was done.

- **Implement a simple discrete event simulator to simulate a queueing network.** Through the use of prototyping a number of incremental versions of a discrete event simulator was created early on in the project, each version being tested for correctness before the next feature was added. A simple discrete event simulator was rapidly created after Christmas and formed the basis for later versions of the software that would use parallel programming libraries.

- **Modifying the simple discrete event simulator to incorporate one parallel discrete event simulation protocol.** As the software development progressed subsequent milestones began to modify to existing simulator to incorporate the conservative PDES protocol. A logical process class was designed and created to represent the LPs from the protocol and a period of testing confirmed the correctness of the design, ironing out any bugs that were found.
Extend the parallel discrete event simulator to run on a Grid test bed. Parallel programming was added to the software through the MPICH library, a lightweight and efficient implementation of the MPI standard, which enabled message passing between individual processes in the software. A Grid enabled version of this implementation called MPICH-G2, that enabled normal MPI programs to run on a Grid over the Globus toolkit, was then used to allow the current software to use a Grid platform for the simulation experiments.

A possible extensions for the project was proposed at the start, which was:

- Extending the simulator to run a large scale wide area network Grid, such as the White Rose Grid.

The length of time it took to run the initial experiments and the complexity of the simulator that was built meant that configuring it to run on a larger WAN Grid would have been an extremely large undertaking. When this possibility was initially raised I heard reports of problems with MPI programs being ran on the WRG. As a result I decided that I didn’t have enough time to investigate this properly and concentrated on creating a fully features PDES with a good experiment, having already completed features that went far past the initial minimum requirements.

Although this initial extension wasn’t completed for various reasons the actual software simulator produced and the series of experiments performed integrated a number of other extensions that went far beyond what was initial planned. Through the use of a solid design stage and rapid prototyping of different software versions, the eventual parallel discrete event simulator that was produced became fairly complex and was far from simple. Advanced features of the asynchronous discrete event simulation protocol were implemented which included deadlock prevention, which was dependent on the creation of null messages in software and also the calculation of the lookahead value, the message arrival code implemented to use a Poisson distribution curve, for more accurate simulation, and the simulators topology was dynamically created at run time, using a custom file format, to reduce the need for major code changes as the experiment changed. A thorough experiment was also devised to properly test the performance of the software in a situation that reflected the real world, as well as mitigating a number of possible sources of error, and the data gathered was then further analysed using a collection of tools, many self written in Python, allowing me to finally draw a series of comprehensive conclusions.
6.2.4 Project Management

In my mid-term report I developed a Gantt chart, figure 6.2.6 in Appendix C, that was an initial outline of the time I planned to spend on each stage of the project, with the stages broken down into further parts. Having never completed a project of this size or scale before a lot of this first Gantt chart involved a significant amount of careful estimation on my part. As the project progressed the actual time commitments required became more obvious and subsequently the chart got revised when appropriate. I deliberately choose an uneven split of modules, doing more in my first semester than my second, because I knew that although I might be slightly behind when it got to Christmas it would allow me large periods of uninterrupted time in the second semester to concentrate on the project. This assumption provided correct and it has been a massive help. Although I was indeed slightly behind on my schedule by the end of the Christmas period I easily recovered this time over the course of the second semester.

I made a conscious effort to start early and do little bits of work often, which was essential considering the amount of new technologies and environments I had to familiarise myself with. This approach paid off and sticking to this approach gave myself plenty of time to deal with the unexpected problems that occurred. Whilst my initial Gantt chart was fairly accurate in some parts I largely underestimated the time needed to complete the project writeup. Luckily however because I’d started early, completing small sections of my writeup as soon as possible, I soon began to realise just how long I would need to allocate at the end for the final write up and modified my plan. My final project schedule is recorded in my revised Gantt chart, figure 6.2.7 in Appendix C.

6.2.5 Environment

The computing environment, which the experiments were run in, was on reflection an appropriate place in which to carry out the tests. The specifications of both machines and the network connections between them was suitable for the type of experiments being completed and I believe did correctly reproduce the conditions found in many real world scenarios. The additional software used in the experiment, including the MPICH libraries and the Globus toolkit, were an excellent choice and again were chosen because of their maturity as software products and wide usage in the real world. Using C++ as the main development language posed no problems throughout the project and many of its specific features, like the Standard Template Library, were used extensively and brought enormous benefits.

Whilst I had no major concerns regarding the environment’s setup for this project, one slight com-
lication did arise and caused some trouble. Both machines used different versions of many of the key software libraries and applications, MPICH, Globus and the C++ compiler. This factor meant that the PDES simulator worked perfectly on one machine but failed when it ran on the other. The source of these problems was sometimes extremely mysterious and usually the appearance intermittent. It’s possible that this combination of differing software versions may have contributed to the serious problems I had with my software crashing sporadically with large messages.

6.2.6 Methodology

The parallel discrete event simulator that I created for this project followed the prototyping methodology during the software development stage. The use of UML in the earlier design stages helped create a solid design for the software application and ease the implementation stage. The rapid prototyping process that is used in creating software with the prototyping methodology meant I was able to build up my knowledge of the different technologies and platforms involved gradually, moving on to the next stage when I was confident, instead of having tackled the final product as a whole. The development methodology chosen was therefore an excellent choice and really benefited my work in the software development stage.
Bibliography


Appendix A - Personal Reflection

Working on a project like this has been extremely enjoyable and looking at what I have produced, through a lot of hard work, I feel extremely happy and proud about my achievements. I found the overall process very worthwhile and on the whole the work I’ve been required to do has been interesting. I chose this project because it managed to tie together two separate areas in computing both that I think looked enjoyable and unfortunately I didn’t have the chance to study these topics in any depth on any of my university modules.

Grid computing is a topic that I have heard a lot about in the computing world recently but have practically no real experience with any of the technologies involved and from what I’d heard it sounded very promising. Parallel computing used to be the subject of a module at Leeds until last year, when it was removed from the list, and one which I’d been looking forward to doing. When I was looking for a possible project and saw this description I jumped at the chance as it allowed me to learn about both Grid computing and parallel programming, having missed out on the opportunity to learn them formally.

I considered a number of other projects before making my final choice and now looking back I believe I definitely made the correct decision. My main overriding belief was that I should choose a project that I would find intellectually stimulating because the project would take up a substantial part of my time in the third year and if I didn’t find the project interesting then it would be hard to motivate myself later on. This would be one of my most essential pieces of advice for any students contemplating which project they should pick. Having chosen your project you’re committed to following it through, the results of which will form a large part of your degree mark for the third year. Picking the right project, something that you find interesting and will challenge you, is one of the most important things
I’ve realised when reflecting on my experiences.

Choosing a project where little or no previous experience in the subject matter is held means that the initial stages are of paramount importance. In my case the project involved working with a whole range of new technologies and platforms which I hadn’t any previous experience with. This meant I had a large amount of independent reading and learning to complete before I could realistically start any practical stages. I realised that this was a crucial stage in the project as it would have a direct impact on my ability to create the simulation software later on. Not skipping on any of the initial research work meant I was fully prepared for the next stages.

Starting early I began to equip myself with the necessary skills and using the prototyping development model I was able begin to put these new skills into practise with simple applications that I would later build on in my final software. Obviously problems occurred but because I’d started early enough I had time to work through them by consulting third parties or using my knowledge. Starting early is of vital importance as it will give any student a better idea of the work load involved and give time to handle any unforeseen problems without the stress of approaching deadlines. Leaving work till later dates will no doubt increase the risk of cutting corners in order to finish on time and could doubly impact the later stages because of this inadequate preparation.

Another enormous help was the input of my assessor and supervisor, through them both I gained a large amounts of help and positive contributions. Karim very importantly stressed the need for constant and regular supervision, usually through the weekly meetings we had for half an hour. This regular feedback and guidance was invaluable and whenever I encountered problems I had one source of information always on hand, either at the meetings or using email. Also on the occasions when I would have communications with my assessor, the mid term report and my demonstration, I was able to discuss any issues I had and gained invaluable advice from his perception. I would advise any student embarking on a final year project to make the most out of these meetings as, having overseen a number of projects each year, their supervisor and assessor are usually the best people to talk to when encountering difficulties or looking to pick up advice about the project.

Using a number of unfamiliar technologies in my project meant that the actual writing of the software could be intensely frustrating at times. On a number of occasions I had problems occur that seemed spontaneous and rather intermittent. In a usual situation, where the software would be for a module coursework, I would be able to seek the advice of fellow classmates and obviously this wasn’t possible. Both my major problems, when the system crashed with large messages and a misplaced
receive call, where the result of assumptions I made in the testing procedures I created for the code. Although starting early meant I had time to spend debugging the problems it impressed on me one aspect of the software construction that is usually overlooked, the testing. Having worked in industry I knew the important role of testing and created a number of unit tests for a number of my prototypes but with later versions I assumed that because the previous prototypes worked correctly, with these forming the basis for later versions, the need for thorough testing was less important. This assumption cost me many valuable hours later on. Another aspect I’d therefore push on any new student starting a project is the important need for comprehensive testing of any software that is constructed, especially in a similar project which has a lot of unknown variables.

Finally the time needed for the project writeup should never be underestimated. Looking at my initial Gantt chart, Figure 6.2.6 in Appendix C, it’s obvious that I thought a few weeks would be sufficient to write up the report, which is over 50 pages in length. Putting in the initial effort with the mid term report and the sample chapter certainly paid off later but I still needed to spend large periods of time on the project write up. Leaving this to the last minute would be extremely stressful and no doubt force the student to produce substandard work.
Appendix B - Experimental Data and Graphs

MPI Results

<table>
<thead>
<tr>
<th>Message count</th>
<th>Run 1</th>
<th>Run 2</th>
<th>Run 3</th>
<th>Mean</th>
<th>Std. Dev.</th>
</tr>
</thead>
<tbody>
<tr>
<td>5000</td>
<td>32.1719</td>
<td>32.168</td>
<td>32.1719</td>
<td>32.1706</td>
<td>0.002252</td>
</tr>
<tr>
<td>10000</td>
<td>64.5312</td>
<td>64.4805</td>
<td>64.4492</td>
<td>64.4869</td>
<td>0.041381</td>
</tr>
<tr>
<td>15000</td>
<td>96.8789</td>
<td>96.8438</td>
<td>96.8477</td>
<td>96.8568</td>
<td>0.019238</td>
</tr>
<tr>
<td>20000</td>
<td>128.926</td>
<td>128.898</td>
<td>128.902</td>
<td>128.907</td>
<td>0.015144</td>
</tr>
<tr>
<td>25000</td>
<td>161.449</td>
<td>161.445</td>
<td>161.441</td>
<td>161.445</td>
<td>0.004000</td>
</tr>
<tr>
<td>30000</td>
<td>193.887</td>
<td>193.867</td>
<td>193.875</td>
<td>193.8763</td>
<td>0.010066</td>
</tr>
<tr>
<td>35000</td>
<td>227.062</td>
<td>227.066</td>
<td>227.062</td>
<td>227.0633</td>
<td>0.002309</td>
</tr>
<tr>
<td>40000</td>
<td>259.496</td>
<td>259.516</td>
<td>259.527</td>
<td>259.513</td>
<td>0.015716</td>
</tr>
<tr>
<td>45000</td>
<td>292.637</td>
<td>292.789</td>
<td>292.641</td>
<td>292.689</td>
<td>0.086626</td>
</tr>
<tr>
<td>50000</td>
<td>324.844</td>
<td>324.84</td>
<td>324.965</td>
<td>324.883</td>
<td>0.071042</td>
</tr>
<tr>
<td>100000</td>
<td>651.18</td>
<td>651.219</td>
<td>651.176</td>
<td>651.1917</td>
<td>0.023756</td>
</tr>
<tr>
<td>250000</td>
<td>1623.19</td>
<td>1623.19</td>
<td>1633.09</td>
<td>1626.49</td>
<td>5.715768</td>
</tr>
<tr>
<td>500000</td>
<td>3240.73</td>
<td>3240.6</td>
<td>3240.69</td>
<td>3240.6733</td>
<td>0.066583</td>
</tr>
</tbody>
</table>

Table 6.1: Simulation time results (seconds)
<table>
<thead>
<tr>
<th>Message count</th>
<th>Run 1</th>
<th>Run 2</th>
<th>Run 3</th>
<th>Mean</th>
</tr>
</thead>
<tbody>
<tr>
<td>5000</td>
<td>11035884</td>
<td>11023984</td>
<td>11024538</td>
<td>11028135</td>
</tr>
<tr>
<td>10000</td>
<td>21808948</td>
<td>21794512</td>
<td>21805034</td>
<td>21802831</td>
</tr>
<tr>
<td>15000</td>
<td>32550936</td>
<td>32548184</td>
<td>32534664</td>
<td>32544594</td>
</tr>
<tr>
<td>20000</td>
<td>43297198</td>
<td>43283558</td>
<td>43272520</td>
<td>43284425</td>
</tr>
<tr>
<td>25000</td>
<td>54001178</td>
<td>54005306</td>
<td>53992966</td>
<td>53999816</td>
</tr>
<tr>
<td>30000</td>
<td>64716286</td>
<td>64747598</td>
<td>64753250</td>
<td>64739044</td>
</tr>
<tr>
<td>35000</td>
<td>75534974</td>
<td>75540568</td>
<td>75526860</td>
<td>75534134</td>
</tr>
<tr>
<td>40000</td>
<td>86266114</td>
<td>86317128</td>
<td>86265186</td>
<td>86282809</td>
</tr>
<tr>
<td>45000</td>
<td>97075772</td>
<td>97069620</td>
<td>97046568</td>
<td>97063986</td>
</tr>
<tr>
<td>50000</td>
<td>107828570</td>
<td>107838964</td>
<td>107858070</td>
<td>107841868</td>
</tr>
<tr>
<td>100000</td>
<td>215324122</td>
<td>215368786</td>
<td>215363958</td>
<td>215352288</td>
</tr>
<tr>
<td>250000</td>
<td>537982724</td>
<td>537932572</td>
<td>538113108</td>
<td>538009468</td>
</tr>
<tr>
<td>500000</td>
<td>1075419440</td>
<td>1075481230</td>
<td>1075427664</td>
<td>1075442778</td>
</tr>
</tbody>
</table>

Table 6.2: Simulation data transfer results (data size in bytes)

<table>
<thead>
<tr>
<th>Message count</th>
<th>Run 1</th>
<th>Run 2</th>
<th>Run 3</th>
<th>Mean</th>
</tr>
</thead>
<tbody>
<tr>
<td>5000</td>
<td>71588</td>
<td>71433</td>
<td>71442</td>
<td>71487</td>
</tr>
<tr>
<td>10000</td>
<td>142542</td>
<td>142370</td>
<td>142502</td>
<td>142471</td>
</tr>
<tr>
<td>15000</td>
<td>213138</td>
<td>213106</td>
<td>212941</td>
<td>213061</td>
</tr>
<tr>
<td>20000</td>
<td>283784</td>
<td>283621</td>
<td>283482</td>
<td>283629</td>
</tr>
<tr>
<td>25000</td>
<td>353898</td>
<td>353963</td>
<td>353811</td>
<td>353890</td>
</tr>
<tr>
<td>30000</td>
<td>424167</td>
<td>424554</td>
<td>424623</td>
<td>424448</td>
</tr>
<tr>
<td>35000</td>
<td>495686</td>
<td>495767</td>
<td>495599</td>
<td>495684</td>
</tr>
<tr>
<td>40000</td>
<td>566154</td>
<td>566774</td>
<td>566143</td>
<td>566357</td>
</tr>
<tr>
<td>45000</td>
<td>637573</td>
<td>637482</td>
<td>637209</td>
<td>637421</td>
</tr>
<tr>
<td>50000</td>
<td>708298</td>
<td>708424</td>
<td>708643</td>
<td>708455</td>
</tr>
</tbody>
</table>

Table 6.3: Simulation data transfer results (number of data packets)
Grid Results

<table>
<thead>
<tr>
<th>Message count</th>
<th>Run 1</th>
<th>Run 2</th>
<th>Run 3</th>
<th>Mean</th>
<th>Std. Dev.</th>
</tr>
</thead>
<tbody>
<tr>
<td>5000</td>
<td>32.1889</td>
<td>32.611</td>
<td>32.1689</td>
<td>32.3229</td>
<td>0.249673</td>
</tr>
<tr>
<td>10000</td>
<td>65.512</td>
<td>64.5816</td>
<td>64.5852</td>
<td>64.8929</td>
<td>0.536130</td>
</tr>
<tr>
<td>15000</td>
<td>96.9615</td>
<td>96.8773</td>
<td>96.8984</td>
<td>96.9124</td>
<td>0.043811</td>
</tr>
<tr>
<td>20000</td>
<td>128.954</td>
<td>129.096</td>
<td>128.993</td>
<td>129.0143</td>
<td>0.073364</td>
</tr>
<tr>
<td>25000</td>
<td>161.489</td>
<td>161.503</td>
<td>163.72</td>
<td>162.2373</td>
<td>1.284046</td>
</tr>
<tr>
<td>30000</td>
<td>196.603</td>
<td>194.002</td>
<td>196.783</td>
<td>195.796</td>
<td>1.556254</td>
</tr>
<tr>
<td>35000</td>
<td>227.142</td>
<td>227.297</td>
<td>227.138</td>
<td>227.1923</td>
<td>0.090666</td>
</tr>
<tr>
<td>40000</td>
<td>259.556</td>
<td>259.585</td>
<td>259.583</td>
<td>259.5746</td>
<td>0.016197</td>
</tr>
<tr>
<td>45000</td>
<td>292.761</td>
<td>292.699</td>
<td>292.703</td>
<td>292.721</td>
<td>0.034699</td>
</tr>
<tr>
<td>50000</td>
<td>329.74</td>
<td>324.916</td>
<td>324.908</td>
<td>326.5213</td>
<td>2.787450</td>
</tr>
<tr>
<td>100000</td>
<td>651.396</td>
<td>651.414</td>
<td>651.219</td>
<td>651.343</td>
<td>0.107764</td>
</tr>
<tr>
<td>250000</td>
<td>1645.75</td>
<td>1623.26</td>
<td>1623.69</td>
<td>1630.9</td>
<td>12.862274</td>
</tr>
<tr>
<td>500000</td>
<td>3246.69</td>
<td>3241.03</td>
<td>3285.33</td>
<td>3257.6833</td>
<td>24.109387</td>
</tr>
</tbody>
</table>

Table 6.4: Simulation time results in milliseconds
Table 6.5: Simulation data transfer results (data size in bytes)

<table>
<thead>
<tr>
<th>Message count</th>
<th>Run 1</th>
<th>Run 2</th>
<th>Run 3</th>
<th>Mean</th>
</tr>
</thead>
<tbody>
<tr>
<td>5000</td>
<td>25466013</td>
<td>25484187</td>
<td>25417424</td>
<td>25455874</td>
</tr>
<tr>
<td>10000</td>
<td>50812765</td>
<td>50677190</td>
<td>50699296</td>
<td>50729750</td>
</tr>
<tr>
<td>15000</td>
<td>75950515</td>
<td>76096245</td>
<td>76367874</td>
<td>76138211</td>
</tr>
<tr>
<td>20000</td>
<td>101369673</td>
<td>101314138</td>
<td>101509645</td>
<td>101397818</td>
</tr>
<tr>
<td>25000</td>
<td>126931214</td>
<td>126520137</td>
<td>126758812</td>
<td>126736721</td>
</tr>
<tr>
<td>30000</td>
<td>15216433</td>
<td>15196975</td>
<td>15239035</td>
<td>152174814</td>
</tr>
<tr>
<td>35000</td>
<td>177486662</td>
<td>17775015</td>
<td>177866160</td>
<td>177702612</td>
</tr>
<tr>
<td>40000</td>
<td>202328077</td>
<td>202402537</td>
<td>203123703</td>
<td>202618105</td>
</tr>
<tr>
<td>45000</td>
<td>228226246</td>
<td>228469869</td>
<td>228281714</td>
<td>228325943</td>
</tr>
<tr>
<td>50000</td>
<td>252983031</td>
<td>253711081</td>
<td>252851268</td>
<td>253181793</td>
</tr>
<tr>
<td>100000</td>
<td>477951257</td>
<td>476401726</td>
<td>469792769</td>
<td>474715250</td>
</tr>
<tr>
<td>250000</td>
<td>1269726535</td>
<td>1268130893</td>
<td>1270811699</td>
<td>1269556375</td>
</tr>
<tr>
<td>500000</td>
<td>2531291778</td>
<td>2525318816</td>
<td>2533737241</td>
<td>2530115945</td>
</tr>
</tbody>
</table>

Table 6.6: Simulation data transfer results (number of data packets)

<table>
<thead>
<tr>
<th>Message count</th>
<th>Run 1</th>
<th>Run 2</th>
<th>Run 3</th>
<th>Mean</th>
</tr>
</thead>
<tbody>
<tr>
<td>5000</td>
<td>151244</td>
<td>150894</td>
<td>150992</td>
<td>151043</td>
</tr>
<tr>
<td>10000</td>
<td>301331</td>
<td>301896</td>
<td>300954</td>
<td>301393</td>
</tr>
<tr>
<td>15000</td>
<td>450220</td>
<td>451711</td>
<td>456272</td>
<td>452734</td>
</tr>
<tr>
<td>20000</td>
<td>603154</td>
<td>602464</td>
<td>603839</td>
<td>603152</td>
</tr>
<tr>
<td>25000</td>
<td>757464</td>
<td>751400</td>
<td>753674</td>
<td>754179</td>
</tr>
<tr>
<td>30000</td>
<td>904906</td>
<td>904388</td>
<td>908400</td>
<td>905898</td>
</tr>
<tr>
<td>35000</td>
<td>1057431</td>
<td>1060352</td>
<td>1061024</td>
<td>1059602</td>
</tr>
<tr>
<td>40000</td>
<td>1201275</td>
<td>1200243</td>
<td>1210108</td>
<td>1203875</td>
</tr>
<tr>
<td>45000</td>
<td>1360815</td>
<td>1362727</td>
<td>1360910</td>
<td>1361484</td>
</tr>
<tr>
<td>50000</td>
<td>1502357</td>
<td>1514632</td>
<td>1500964</td>
<td>1505984</td>
</tr>
</tbody>
</table>
Figure 6.1: Graph plotting the mean simulation times for 5,000 to 50,000 messages, using the MPI and Grid platforms, and the perfect throughout of 200 messages a second.
Figure 6.2: Graph plotting the mean data transfers for 5,000 to 50,000 messages, using the MPI and Grid platforms.

Figure 6.3: Graph plotting the number of mean network packets for 5,000 to 50,000 messages, using the MPI and Grid platforms.
Figure 6.4: Graph plotting the mean simulation times for 5,000 to 500,000 messages, using the MPI and Grid platforms, and the perfect throughout of 200 messages a second (logarithmic axis scaling).

Figure 6.5: Graph plotting the mean data transfers for 5,000 to 500,000 messages, using the MPI and Grid platforms (logarithmic axis scaling).
Appendix C - Project Schedule Charts

Figure 6.6: Original project schedule, Gantt chart

Figure 6.7: Revised project schedule, Gantt chart
Appendix D - UML Design Diagrams

Figure 6.8: Original class diagram for the software
Figure 6.9: Final class diagram for the software

Figure 6.10: Sequence Diagram, showing the main program logic for the software simulator