A SIMULATION KERNEL

JAMES ADAMS

BACHELOR OF SCIENCE IN COMPUTER SCIENCE WITH HONOURS

UNIVERSITY OF BATH

MAY 2007
CONSULTATION RELEASE

This dissertation may be made available for consultation within the University Library and may be photocopied or lent to other libraries for the purposes of consultation.

SIGNED:
A SIMULATION KERNEL

SUBMITTED BY: JAMES ADAMS

COPYRIGHT

Attention is drawn to the fact that copyright of this dissertation rests with its author. The Intellectual Property Rights of the products produced as part of the project belong to the University of Bath (see http://www.bath.ac.uk/ordinances/#intelprop).

This copy of the dissertation has been supplied on condition that anyone who consults it is understood to recognize that its copyright rests with its author and that no quotation from the dissertation and no information derived from it may be published without the prior written consent of the author.

DECLARATION

This dissertation is submitted to the University of Bath in accordance with the requirements of the degree of Bachelor of Science in the Department of Computer Science. No portion of the work in this dissertation has been submitted in support of an application for any other degree or qualification of this or any other university or institution of learning. Except where specifically acknowledged, it is the work of the author.

SIGNED:
ABSTRACT

Parallel Discrete Event Simulation (PDES), is the execution of a single discrete event simulation across multiple processors. The CCT algorithm has proven an effective approach with which to execute large scale, low event granularity communication network simulations.

This project aims to produce a simulation kernel based on the CCT algorithm, built from the ground up with an emphasis on efficiency, usability, simplicity, and extensibility. An iterative approach is used to build up layers of functionality and two simulation models are developed to collect performance results in various customized simulation scenarios.

Significant speed up is achieved, particularly in large scale, and high event granularity models.
# Table of Contents

1 Introduction ...................................................................................................................... 1  
  1.1 Project Aim .................................................................................................................. 1  
  1.2 Project Objectives ....................................................................................................... 1  
  1.3 Document Structure ................................................................................................. 1  
  1.4 Simulation .................................................................................................................. 2  
2 Literature Survey .............................................................................................................. 6  
  2.1 Introduction ................................................................................................................ 6  
  2.2 Simulation .................................................................................................................. 6  
  2.3 Computer Simulation ............................................................................................... 7  
  2.4 Parallel Simulation Development ............................................................................ 8  
  2.5 Can the Field Survive? ............................................................................................. 17  
  2.6 Topic Summaries and Discussions ........................................................................... 19  
  2.7 Problems Encountered ............................................................................................ 20  
  2.8 Computer Simulation Languages ............................................................................ 22  
  2.9 Application Areas ..................................................................................................... 23  
  2.10 Conclusion .............................................................................................................. 24  
3 The CCT Algorithm .......................................................................................................... 25  
  3.1 Introduction ................................................................................................................ 25  
  3.2 The CCT Algorithm .................................................................................................. 28  
4 Parallel Simulation Analysis ............................................................................................ 30  
  4.1 Parallel Architectures .............................................................................................. 30  
  4.2 Parallel Overhead ...................................................................................................... 30  
  4.3 Amdahl's Law ........................................................................................................... 31  
  4.4 PDES Performance Factors ..................................................................................... 32  
5 Requirements .................................................................................................................... 37  
  5.1 Functional Requirements ......................................................................................... 37  
  5.2 Design Criteria .......................................................................................................... 37  
6 Design & Implementation ................................................................................................. 40  
  6.1 Design....................................................................................................................... 40  
  6.2 Model Design ............................................................................................................ 57  
7 Testing.............................................................................................................................. 63  
  7.1 Source LP Execution ............................................................................................... 63  
  7.2 Non-Source LP Execution ...................................................................................... 63  
  7.3 Scheduling Internal Critical Channel ..................................................................... 64  
  7.4 Scheduling External Critical Channel ..................................................................... 64  
  7.5 Executing Events Up To The Safe Time ................................................................. 64  
  7.6 Receiving Events From Channels ......................................................................... 64
A Simulation Kernel

7.7 Discarding Events Past The Simulation End Time .......................................................... 65
7.8 Self Scheduled Events ........................................................................................................ 65
8 Results .................................................................................................................................. 66
  8.1 Observed Speedup .............................................................................................................. 66
  8.2 Simulation Scale ............................................................................................................... 67
  8.3 Event Granularity ............................................................................................................ 70
  8.4 Time Parallelism ............................................................................................................... 75
  8.5 Thread Blocking .............................................................................................................. 75
  8.6 Memory Usage ................................................................................................................ 77
  8.7 Processor Time ................................................................................................................ 79
9 Evaluation & Conclusion ..................................................................................................... 81
  9.1 Project Aims ..................................................................................................................... 81
  9.2 Results ............................................................................................................................ 81
  9.3 Improving Performance ................................................................................................... 82
  9.4 Requirements and Design .............................................................................................. 84
  9.5 Implementation ............................................................................................................... 85
  9.6 Possible Extensions ......................................................................................................... 87
10 Bibliography ....................................................................................................................... 88
Appendix A: Testing Log Outputs .......................................................................................... 95
Appendix B: Results Tables ................................................................................................... 100
Appendix C: Code Snippets .................................................................................................... 102
Appendix D: Running Instructions .......................................................................................... 103
# Table of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Figure 1.1</td>
<td>Specified Application Areas of the World’s Top 500 Supercomputers [2]</td>
<td>3</td>
</tr>
<tr>
<td>Figure 2.1</td>
<td>A program comprising of two tasks; Task A which takes approximately 75% of the execution time, and Task B which takes the remaining 25%. It is clear that executing Task A twice as fast gives a substantially better result than executing Task B five times as fast.</td>
<td>32</td>
</tr>
<tr>
<td>Figure 4.1</td>
<td>Amdahl’s Law for Parallel Speedup</td>
<td>31</td>
</tr>
<tr>
<td>Figure 4.2</td>
<td>An Example Task Execution Pattern, with Three Tasks Waiting for a Fourth to Complete</td>
<td>34</td>
</tr>
<tr>
<td>Figure 6.1</td>
<td>The basic, high level system flow. A simulation model is the input, and the simulation results, and logging information are the outputs.</td>
<td>41</td>
</tr>
<tr>
<td>Figure 6.2</td>
<td>Initial class diagram showing basic classes and fields specified in the algorithm definition</td>
<td>42</td>
</tr>
<tr>
<td>Figure 6.3</td>
<td>Class diagram showing base algorithm framework methods and data structures.</td>
<td>45</td>
</tr>
<tr>
<td>Figure 6.4</td>
<td>Class diagram showing base algorithm and simulation model construction methods and data structures</td>
<td>51</td>
</tr>
<tr>
<td>Figure 6.5</td>
<td>The Logical Process abstract class diagram</td>
<td>55</td>
</tr>
<tr>
<td>Figure 6.6</td>
<td>The Channel class diagram</td>
<td>55</td>
</tr>
<tr>
<td>Figure 6.7</td>
<td>The Task class, which implements the Runnable interface</td>
<td>56</td>
</tr>
<tr>
<td>Figure 6.8</td>
<td>The Simulation abstract class diagram</td>
<td>56</td>
</tr>
<tr>
<td>Figure 6.9</td>
<td>The EventComparator and TaskComparator classes which implement the Comparator interface.</td>
<td>56</td>
</tr>
<tr>
<td>Figure 6.10</td>
<td>An example Pipe Simulation LP topology</td>
<td>58</td>
</tr>
<tr>
<td>Figure 6.11</td>
<td>The class diagram for the Pipe Simulation model</td>
<td>59</td>
</tr>
<tr>
<td>Figure 6.12</td>
<td>The general topology used in the request response simulation model</td>
<td>62</td>
</tr>
<tr>
<td>Figure 8.1</td>
<td>Speedup Graph for Small Request Response Model</td>
<td>68</td>
</tr>
<tr>
<td>Figure 8.2</td>
<td>Speedup Graph for Medium Request Response Model</td>
<td>69</td>
</tr>
<tr>
<td>Figure 8.3</td>
<td>Speedup Graph for Medium Request Response Model</td>
<td>70</td>
</tr>
<tr>
<td>Figure 8.4</td>
<td>Speedup graph for very low granularity events</td>
<td>71</td>
</tr>
<tr>
<td>Figure 8.5</td>
<td>Speedup graph for low granularity events</td>
<td>72</td>
</tr>
<tr>
<td>Figure 8.6</td>
<td>Speedup graph for high granularity event test</td>
<td>73</td>
</tr>
<tr>
<td>Figure 8.7</td>
<td>Speedup graph for very high granularity event test</td>
<td>74</td>
</tr>
<tr>
<td>Figure 8.8</td>
<td>Event granularity speedup comparison graph</td>
<td>74</td>
</tr>
<tr>
<td>Figure 8.9</td>
<td>Time parallelism speedup results graph</td>
<td>75</td>
</tr>
<tr>
<td>Figure 8.10</td>
<td>JProfiler thread monitor graph. [Yellow] = Sleep, [Red] = Blocked.</td>
<td>75</td>
</tr>
<tr>
<td>Figure 8.11</td>
<td>Processor utilisation graph for thread blocking analysis</td>
<td>77</td>
</tr>
<tr>
<td>Figure 8.12</td>
<td>JVM memory usage during a large scale simulation run</td>
<td>78</td>
</tr>
<tr>
<td>Figure 8.13</td>
<td>JVM memory usage during a small scale simulation run</td>
<td>79</td>
</tr>
<tr>
<td>Figure 8.14</td>
<td>Processor time hot spots in the simulation kernel.</td>
<td>80</td>
</tr>
</tbody>
</table>
# Table of Tables

| Table B.1 | Small Request Response Model Results | ................................................................. | 100 |
| Table B.2 | Medium Request Response Model Results | ........................................................................... | 100 |
| Table B.3 | Large Request Response Model Results  | ........................................................................... | 100 |
| Table B.4 | Very low event granularity test results | ........................................................................... | 101 |
| Table B.5 | Low event granularity test results     | ........................................................................... | 101 |
| Table B.6 | High event granularity test results    | ........................................................................... | 101 |
| Table B.7 | Very high event granularity test results | ........................................................................... | 101 |
ACKNOWLEDGEMENTS

I would like to thank my project supervisor Dr. Russell Bradford for his guidance during the project. I would also like to thank my family and friends for their support.
1 INTRODUCTION

1.1 PROJECT AIM

The main aim of this project is to develop a parallel simulation kernel based on the Critical Channel Traversal (CCT) algorithm first proposed by Xiao et al [1].

1.2 PROJECT OBJECTIVES

In order to meet the project aim, several objectives need to be achieved:

- Design and implement a simulation kernel.
  - Using at its core the scheduling mechanisms defined in [1]
  - Capable of executing any suitable simulation model
  - Capable of effectively utilising multiple processors when available

- Provide evidence for the correctness of the simulation kernel in relation to the definition of the CCT algorithm by executing test case simulation models

- Collect performance results for the simulation kernel in various simulation scenarios and across variable processor counts

1.3 DOCUMENT STRUCTURE

This chapter serves as an introduction to the project, listing the aims and objectives of the project, and providing a high-level description of the problem domain. Section 1.4.4 describes briefly why parallel simulation is difficult, and shows that a new approach from the traditional sequential execution model is required.

Chapter 2 contains a survey of past literature on the topic of computer simulation, and in particular that of parallel simulation. Different approaches to simulation are discussed, and the CCT algorithm is given context as a relatively recent conservative approach.

Chapter 3 describes the CCT algorithm itself, and provides a simplified overview of the scheduling and event execution mechanisms used. Formal definitions are given for the various objects introduced, and these are used from this section on.
Chapter 4 further explores some of the key topics in parallel computing and parallel simulation that were introduced in the literature survey. An understanding of these topics is important to ensure that informed decisions can be made during the design and implementation phases of the project.

Chapter 5 defines the functional requirements of the project, as well as a number of design principles which were used to guide decisions required in the design and implementation phases.

Chapter 6 documents the design and implementation stages of the project. The chapter starts with the reasoning behind the choice of language for the project, and continues through three iterations of development. The three iterations address the need for separate areas of functionality; the core CCT algorithm, execution scheduling, and model development. This chapter end with the design of two simulation models.

Chapter 7 contains testing results and discussion in an effort to prove the correctness of the CCT algorithm implementation. Output logs in Appendix A are referenced as evidence of system behaviour in various test cases.

Chapter 8 contains the performance results of testing the simulation kernel using two different simulation models. Performance results are collected with varying processor counts across the following variables; event granularity, simulation scale, and time parallelism potential.

Chapter 9 evaluates the project outcome against the original project aims, critiquing the results obtained, and the processes used throughout. Some possible future work is outlined.

1.4 SIMULATION

Simulation is the technique of imitating the behaviour of a system by building an analogous model of that system for the purposes of testing, information gathering or personnel training. Simulations come in many forms across a vast array of application domains; model cockpit flight simulators used to train airline and fighter pilots; economic models used by financial institutions for scenario and risk planning; network traffic analysis, and many more. Figure 1.1 shows the specified application domain uses of the current top 500 super computers.
In this project we are interested in the running of simulations on computers which consist primarily of two parts; a computer based model of the real object or system, and some mechanism or kernel for ‘running’ the model.

1.4.1 Parallel Simulation

Increased commercial demand has driven the need for increasingly complex simulations, and the technology and methodologies for running these simulations has failed to keep up. The performance of sequential simulations using traditional mechanisms, even on the fastest machines, does not meet the demands being placed upon them. Communication systems, such as mobile phone networks, have demonstrated a need for analytical tools for which mathematical modelling and sequential simulation have proved inadequate.

Parallel simulation kernels attempt to exploit the potential for parallelism in simulation problems by utilising multiprocessor architectures. Unfortunately, conventional centralised event-list driven techniques cannot be used on multiprocessor architectures - the high frequency of event list manipulation required does not scale well and quickly becomes a bottleneck [3]. In response to this, a new approach was developed; decomposing simulations into a set of concurrently executing processes, each capable of independently advancing the part of the simulation they model.

The focus of this project is to create a simulation kernel for executing one instance of a simulation across multiple processors, it is important to differentiate this from the concept of executing multiple simulation instances concurrently across multiple processors. This latter form of parallelism can be useful for repeating simulation runs for gathering empirical evidence.
or performing parameter sweeps, but requires more memory, does not speed up execution of
individual simulation runs, and is not suitable for chained simulations where the results of the
previous run affect the next.

Efficient PDES algorithms are particularly important in running simulations of high volume, low
processing overhead systems such as computer networks and other communication systems.
Simulation of these systems does not allow large segments of computation to be allocated and
processed in parallel, but instead relies on high event throughput, achieved through minimal
synchronisation overheads between processes. Network simulations are often used to
benchmark parallel simulation kernels for this reason.

1.4.2 Discrete Event Simulation

Discrete Event Simulation (DES) is a method for building models of time-based systems. The
operation of such systems is represented as a chronological sequence of events. Discrete event
simulations are based on the concept of simulated time combined with an event queue; events
from the event queue are processed in chronological order, with the simulated time being
advanced in hops to the time of the currently executing event. This paradigm transitions well to
parallel architectures, each executing process can maintain a local event queue and process
events from that queue, but unfortunately introduces new difficulties.

1.4.3 Parallel Discrete Event Simulation

Most current PDES strategies break up the system being modelled, often called the physical
system, into the individual physical processes, which interact at various stages to form the
complete system, an idea first proposed in [4]. The simulator is then constructed as a set of
Logical Processes (LPs) which map onto these physical processes. Interactions between physical
processes are modelled as time stamped event messages which are passed between the
corresponding LPs in the simulation.

1.4.4 Why is PDES Hard?

Adapted from Fujimoto’s Introduction to the Difficulties of PDES in [5].

The difficulties of PDES are more readily identified when examining the implicit constraints
provided by sequential discrete event simulators, and the problem of translating these for use in
parallel execution. Consider the standard sequential simulator which uses three main data
structures:

1. State variables, which describe the state of the simulation

2. An event list, which contains all pending events which have been scheduled, but not yet
executed
A Simulation Kernel

3. *A global clock*, to keep track of the point in simulated time the simulation has reached.

Each event has an associated timestamp which denotes the time that the event is scheduled to occur. The simulation proceeds by executing events in the event list in order of increasing timestamp value. Processing an event can change the state variables, and may create new events scheduled in the future.

It is essential that the smallest time stamped event ($E_{\text{min}}$) in the queue be executed each time, any other event could modify the state variables used by $E_{\text{min}}$ and would effectively simulate a system in which the future could affect the past. This is clearly unacceptable, and errors of this nature are called *causality errors*.

Any cause and effect relationships in the system being modelled become sequencing constraints necessary to avoid causality errors in the simulation, and it is the simulation mechanisms responsibility to ensure that these sequencing constraints are not violated.
2 LITERATURE SURVEY

2.1 INTRODUCTION

The field of computer simulation is a broad one, it is necessary to reach some understanding of past research and developments to give context to this project. The research material available is immense and stretches over 30 years. This survey aims to cover the core developments and give context to the Critical Channel Traversal (CCT) algorithm, as well as discuss some of the more general topic areas of Parallel Discrete Event Simulation (PDES).

Two main approaches to PDES are discussed, along with optimisations for each. Alternative approaches are also mentioned.

The history of PDES includes a period of reflection, and some of the concerns and suggestions key proponents in the field have raised are discussed. Several summaries of past research are presented for further reference.

Some of the general problems encountered in parallel simulation research are discussed, as well as their potential solutions when available. This is followed by a description of the role of simulation languages, in particular the languages MEX and MAISIE.

2.2 SIMULATION

2.2.1 WHAT IS SIMULATION?

Many definitions of simulation are available and in a broad form simulation can be thought of as “an imitation of some real thing, state of affairs, or process” [6], however, in the context of this project a more precise definition is useful since we are talking specifically about computer based simulation.

“Simulation is the process of designing a model of a real or imagined system and conducting experiments with that model. The purpose of simulation experiments is to understand the behaviour of the system or evaluate strategies for the operation of the system. Assumptions are made about this system and mathematical algorithms and relationships are derived to describe these assumptions - this constitutes a "model" that can reveal how the system works.” [7]
2.2.2 How are Simulations Performed?

There are two main approaches to performing simulations:

- Physical simulation, where physical objects are substituted for the real object or system, often because they are smaller or cheaper.

- Computer simulation, where a computer based model of the real object or system is built and some mechanism for ‘running’ the model is used. The rest of this document is concerned with computer simulation.

2.3 Computer Simulation

Computer simulation has developed hand-in-hand with the growth and development of the computer. The first large-scale computer simulation was used in the Manhattan Project during World War II to model neutrons and aided development of the first atomic bomb [8, 9].

There is an important distinction between what is being simulated, i.e. the model of the physical process, and how the simulation is being performed by means of some mechanism or ‘kernel’. In the ‘60’s, when computer simulation was drawing more attention, research was being published which established ‘guides’ or frameworks for the development models and simulation mechanisms. Models were categorized as either continuous or discrete, and methods for simulating continuous change models on digital computers were proposed [10]. Simulation was shown as a tool for both helping to build systems through component synthesis, and analyse whole systems so as to decompose them and develop understanding of their components [11].

2.3.1 Discrete Event Simulation

In discrete event simulation, the operation of a system is represented as a chronological sequence of events. Each event occurs at an instant in time and marks a change of state in the system [12]. The time between these instants can be arbitrarily small, and as such is able to model real world interactions.

Sequential discrete event simulators typically contain the following components:

- A Clock: keeps track of the current simulation time

- An Event List: maintains a list of events which need to be processed

- Simulation Engine Logic; processes the events in the simulation. This is also known as the simulation kernel and is the engine by which a simulation is run and as such is the part most open to improvement and posing the most difficult challenges.
2.3.2 The Need for Faster Simulation

Since simulations were possible, researchers have wanted to create more and more complex simulations. This is partly due to the increasing complexity of that which is being modelled as well as a demand for more detailed, and accurate, simulations.

“While the speed of sequential computers continues to increase every year, these increments have been far outstripped by our ability to find new applications that utilize all of the CPU cycles that are available, and then some.” [13]

The required scale of simulations demands more than simply faster processing, the size of some simulations simply outstrips the available memory in single processor architectures. Developing efficient, scalable mechanisms for parallel simulation will enable supercomputers, and distributed processing networks with enormous memory capacity, to be effectively deployed running simulations.

2.3.3 Sequential Simulation Can’t Cope

The performance of sequential simulations using traditional mechanisms, even on the fastest machines, does not meet the demands being placed upon them. Communication systems have demonstrated a need for analytical tools for which mathematical modelling and sequential simulation have proved inadequate. Unfortunately the conventional event-list driven techniques cannot be used on multiprocessor architectures - the high frequency of manipulation required cannot scale well and soon becomes a bottleneck in the system [3].

2.4 Parallel Simulation Development

2.4.1 Logical Processes and Message Passing

Nearly all approaches to Parallel Discrete Event Simulation (PDES) are based around the central concepts of Logical Processes (LPs) and message passing. An LP represents a component of the physical system being modelled and encapsulates that components’ behaviour and state; it also maintains an internal representation of the time that LP has progressed to in the simulation.

Interactions between components in the physical system are modelled using time-stamped event messages passed between the corresponding LPs. Modelling based on message passing proved to be a useful concept, as, ten years after its conception, a language fragment called MAY [4] implemented a message based approach, albeit for sequential algorithms, in an attempt to more naturally model distributed systems.
In splitting the operation of a system into multiple processes, and executing each process on a separate processor it is possible to exploit the modularity and concurrency of the original system in the simulation.

2.4.2 Conservative Approaches

2.4.2.1 Initial Development

Although developed independently, the first algorithms for PDES are very similar and require that the links between communicating processes must be statically specified before execution begins.

In 1977 Bryant [14] published a technical report proposing that a correct simulation (one which models the time and input-output behaviour of a system) could be built using the mechanisms described above. Importantly, Bryant proposed a time-independent algorithm which would allow the simulated behaviour of each component to be independent of the speed of the overall simulation. Bryant also proposed that additional co-ordination operations necessary to prevent deadlock and ensure the simulation termination could be provided without requiring any centralized controlling mechanism.

In 1979 Chandy and Misra published a paper [15] proposing a distributed algorithm for discrete event simulation with similar characteristics to that of Bryant’s; Chandy and Misra had also proved that deadlock could not occur and that no central controlling algorithm was necessary. Bryant’s 1977 thesis is given special mention as being similar to their work, which they state they were not aware of during the development of their algorithm. Chandy and Misra’s proposal is differentiated from similar distributed algorithms [14, 16, 17] in that it proves a bounded memory requirement of each LP. Since any real implementation would be on a memory-limited machine this is vitally important.

Both approaches require the use of null messages to avoid deadlock. Null messages do not relate to any real process or message but are purely a synchronisation mechanism. Whenever an event is processed by an LP, the LP sends null messages containing a time-stamp of the event just processed to all other LPs for which an output link is defined. A process receiving a null message can therefore determine a time up to which it is safe to process events without introducing causality errors (see section 2.7.4).

2.4.2.2 The CMB Algorithm

The framework of parallel discrete event simulation as proposed in [14] and [15] of distributed LPs communicating via message passing, avoiding deadlock and without central control has commonly been referred to as the CMB (Chandy-Misra-Bryant) algorithm or the Chandy-Misra deadlock avoidance algorithm. It was the starting point for conservative parallel simulation, upon which much future work is based, including the CCT algorithm presented later.
2.4.2.2.1 Reducing Null Messages

In 1990 DeVries [18] described the fact that the CMB algorithm often generates a large number of null messages as a major problem, and presents a method for reducing this number based on predicting channel times. DeVries’ improvements can be effective in almost all kinds of LP arrangements and LP event queue types.

Another technique for reducing null messages by sending them on demand rather than after every event is proposed in [19]. In this mechanism an LP which runs out of events it can process is able to poll its upstream LPs, which in return reply with a time stamped null message. This approach succeeds in reducing the number of null messages sent, but seems to introduce a higher latency between components as it requires two messages to be sent.

2.4.2.3 Deadlock Detection and Recovery

In 1981 Chandy and Misra [3] presented another system for parallel discrete event simulation which applies a solution structure, known as a sequence of parallel computations. It differs from previous algorithms in that, rather than deadlock avoidance, it relies on deadlock detection to determine when each phase of parallel computation has ended, and a rollback mechanism to synchronise the processes again.

Chandy and Misra note that it seems more efficient to detect and recover from deadlock than to prevent it occurring. They also improve on the memory bounds of their previous algorithm [15] by proving that the memory required by all processors is at most that required by a single processor executing the simulation sequentially. This allows parallel simulations to be implemented within the memory limits of the computers at the time and, since simulations are often limited by available memory, it also showed that parallel simulation can be a practical tool; one that does not require a vastly expensive computer to run on. Formal proofs are given for the correctness of their algorithm.

Deadlock detection has since been improved, and two alternative methods are described in [20] and [19].

2.4.2.4 Synchronous Operation

Synchronous algorithms [21, 22] proceed in a series of ‘rounds’, in which a set of safe to compute events are determined and processed. Barrier synchronisations are performed to avoid interference between rounds and between components within a round. One disadvantage of synchronous algorithms versus deadlock detection and recovery approaches is that they are prone to unnecessary synchronisation and waiting to prevent deadlocks which might never occur.
2.4.2.5 Time Windows

Time windows were used by Lubachevsky in [21] as an effective mechanism for reducing the search space required to determine whether events were safe to compute or not. The size of the window that can be used is application specific.

2.4.2.6 Critical Path & Critical Time

Jefferson and Reiher [23] investigate the notions of the critical path and critical time of an event, which can be used to give a lower bound on the possible execution time using a conservative algorithms.

The paper discusses optimistic algorithms which are able to beat the critical path bound and describe these mechanisms as capable of supercritical speedup, noting that although this has shown to be possible using optimized test cases, the likelihood of any real-life simulation achieving this for more than very short periods of time is low.

The authors conclude that although a critical path analysis can provide a lower-bound for conservative PDES mechanisms, it still remains an open problem to solve the general case. Critical paths have continued to be an active area of research, and a variant forms the key scheduling mechanism in the CCT algorithm.

2.4.2.7 Critical Channel Traversal

Critical Channel Traversal (CCT) builds on the conservative CMB algorithm and is aimed at simulating network models on shared memory architectures. It was motivated by problems encountered using existing PDES mechanisms on the ATM-TN simulator. CCT uses a multi-level scheduling algorithm that supports scheduling large grains of computation even with low granularity events. Performance is enhanced by ensuring good cache behaviour, automatic load balancing and a limited form of time parallelism [24] - an approach also used in optimistic algorithms.

The CMB algorithm is extended by the addition of rules which determine when LPs should be scheduled based on the number of events waiting to be processed on each LP. This is done through the identification of critical channels in a network. The LP paradigm is extended by defining unidirectional channels which connect a sender and receiver LP.

Logical processes which frequently communicate together and which access the same memory locations are grouped into tasks - in network simulations chains of LPs are common and provide a perfect example of this. Tasks can then be scheduled via a central mechanism which provides a three-tier scheduling system:

1. Tasks
2. Logical processes within a task
3. Events waiting at a logical process.

An implementation of the CCT algorithm named TasKit is available and its performance is comparable to the optimistic Warp Kit [25] and conservative WaiKit. The CCT algorithm has been used for several IP network simulations including [26], and was modified for a real-time IP network simulation in [27].

A more detailed description of the CCT algorithm is given in section 3, on page 25.

2.4.2.7.1 Buffer Exhaustion in CCT

The CCT algorithm is prone to buffer exhaustion in the following situations [28]:

- **Source Vertex Problem**: This problem occurs whenever a source LP continually schedules events for a receiving LP at a rate faster than the receiving LP can handle. As the simulation progresses the event buffer between the two LPs will grow unchecked, and may lead to buffer exhaustion.

- **Source Sub-Graph Problem**: This problem is similar to the above Source Vertex Problem but can occur in systems in which no source LPs are present. Imagine a system such as in Figure 2.1 in which LPs A, B and C form a cycle, with another LP (D) connected to C. If A, B, and C continue to execute normally while D stays idle or executes slowly then the event buffer between LP C and D will grow unchecked.

![Figure 2.1 Three Cyclic LPs with one Hanging LP.](image)

2.4.2.7.2 Improvements to CCT

Two improved versions of the above CCT algorithm are presented in [29] and aim to simplify the original algorithm and provide a better scaling mechanism.
A Simulation Kernel

The *simple sender side* (SSS) CCT algorithm removes the overhead cased by busy waiting in the original CCT algorithm. This has the secondary effect of improving performance in cases where significant slowdown was caused by the inclusion of this mechanism, some of these cases are discussed in [30].

A second improvement proposed in the *receive side* (RS) CCT algorithm changes the way channel clocks are updated and aims to avoid checking channels which are not relevant to scheduling calculations. This has the effect of improving scalability within highly connected models but introduces a slightly greater overhead in sparsely connected models.

Three new versions of CCT are proposed in [28] which substantially reduce the buffer size required for a simulation. The solutions all share a *back schedule request* mechanism which allows out of normal order execution of LPs, based on the number of events currently buffered for an LP.

2.4.3 Optimistic Approaches

2.4.3.1 Virtual Time & Time Warp

Virtual time [31], proposed in 1985, was a new paradigm for organizing and scheduling distributed systems and could be applied to discrete event simulation and database concurrency control. It proposed a more general form of the mechanisms similar to the blocking and distributed deadlock detection given by [3] as well as other previous distributed simulation methods [15, 17, 32, 33].

“*A virtual time system is a distributed system executing in coordination with an imaginary virtual clock that ticks virtual time.*” [31]

Previous distributed systems used some kind of *block-resume* mechanism to keep processes synchronised where rollback was present in the form of transactions and an *abortion-retry* mechanism. In contrast, Virtual Time allows arbitrary *look ahead-rollback* where each process can execute without regard to synchronisation conflicts which are detected and rolled back transparently to the user.

Time Warp has an interesting property not shared with conservative algorithms. It can take advantage of concurrency in situations where an event *could* be sent from one LP to another, but in fact no event is sent.

Virtual Time has two fundamental rules which coincide exactly with Lamport’s Clock Conditions [34]:

1. The virtual time of each message must be less than its virtual receive time
2. The virtual time of each event in a process must be less than the virtual time of the next event at that process.

An implementation of virtual time is provided called the *Time Warp mechanism* which is recognized as the first implementation of an optimistic discrete event simulator [35].

### 2.4.3.1.1 Problems with TimeWarp

TimeWarp introduces two new problems, described in [36], known as *cascading rollbacks* and processor *thrashing*. A cascading rollback, as its name suggests, is when the rollback of one logical process generates additional rollback messages, which in turn generate more, continuing until all processing affected by the initial rollback have been rolled back. This can be a serious problem. Imagine the exaggerated scenario of LP\(_A\) processing one event which takes a minute, the other LPs in the simulation may have processed many events by this point, and advanced their clocks much further than LP\(_A\). If LP\(_A\) then sends a message to these LPs which requires them to rollback to this first messages timestamp all of the work they had done will be lost. In this way it is possible for every processor to be held up by a single slow one.

Processor *thrashing* occurs when a processor is able to rapidly advance its simulated time ahead of other parts of the simulation, usually due to an imbalanced partitioning of LPs to processors but there are many possible causes. When a single process advances so fast there is a greatly increased chance of cascading rollbacks occurring as the process receives messages from other processes catching up.

### 2.4.3.1.2 Special Purpose Hardware

The design and performance of special purpose hardware for Time Warp, in the form of a ‘rollback chip’ which can efficiently implement state saving and rollback functions, is given in [37]. Results of simulation studies show that the use of the rollback ship can virtually eliminate the state saving overhead which plagues software Time Warp implementation. Unfortunately, special purpose hardware will likely only be used in expensive, dedicated simulation computers and the benefits they provide will be unavailable to the majority of the simulation community.

### 2.4.3.1.3 Aggressive and Lazy Cancellation

Two rollback mechanisms for parallel simulation mechanisms based on Time Warp are given in [38]; *aggressive cancellation* in which anti-messages are immediately sent upon rollback, and *lazy cancellation* in which damage from incorrect computations is repaired rather than the computations being repeated.

Lazy cancellation relies on many of the properties which look ahead also exploits; it has the advantage however that it can exploit these properties if they are inherent in a models design –
they do not have to be explicitly programmed for. Lazy cancellation relies on a method of repair being available, which in some situations may not be possible.

2.4.3.1.4 **Rollback Relaxation**

Another optimisation for Time Warp mechanism is proposed in [39]. It eliminates the need to rollback a processes which satisfy the constraint of their output being completely dependent on their input, at all times. It is claimed that 74% of real-life processes qualify for this optimization, however in the network models, which are ideal for stress testing and benchmarking PDES algorithms [40], few LPs satisfy the constraint. This is an application specific optimization which would require tailoring of the simulation algorithm for specific models.

2.4.3.2 **Preventing Error Propagation**

It was shown to be possible to limit the spread of erroneous computations by the use of special control messages in the Wolf algorithm [41]. The algorithm is slightly strange in that it introduces a conservative mechanism, which has the potential of unnecessarily pausing computation, into a generally optimistic solution. The Wolf algorithm can be seen as an intermediate step between optimistic algorithms, and the ANR class of algorithms described in Section 2.4.4.3.

Another simple and effective mechanism for optimizing the cancellation of incorrect computations was proposed by Fujimoto [42] and involves storing a pointer to sent messages, this pointer can then be used if the sent event must be cancelled. This approach both increases the speed of which events can be cancelled, and reduces the overhead of doing so, albeit with a slightly increased memory footprint.

2.4.3.3 **Time Parallelism**

Chandy and Sherman proposed a method for computing the values of state variables across time [24]. A simulation can then be viewed as a two dimensional ‘space-time’ graph which is calculated by the simulation as it progresses. Time parallelism exploits some of the same properties which lazy cancellation relies on.

2.4.3.4 **Reverse Computation**

It is possible to remove the need for periodic state saving in optimistic algorithms by using reverse computation [43]. Reverse computation can recover previous state values by performing a perfect inverse of event computation – the original value is obtained by running the code backwards. The number of models for which reverse computation is possible limits the application of this optimisation.
2.4.3.5 Approximate Time

Approximate time [44] relaxes traditional notion of timestamps by allowing a range instead of a discrete value. This is made possible by an implicit temporal uncertainty in some simulation models (such as the time taken for a vehicle to move between two points in a real-life scenario). This relaxation of timestamps allows flexibility in the way events can be scheduled, especially in conditions required for look ahead.

2.4.4 Other Approaches

Conservative and optimistic approaches have been presented, and the continuing trends for research are mostly in the areas of improving these mechanisms. Reynolds challenges this dichotomy in conventional wisdom in his 1988 paper [35] by describing a design space for parallel simulation which includes current approaches and also implies the existence of many as yet unexplored approaches.

2.4.4.1 Look Back

In [45] Chen et al propose a new class of PDES called look back-based protocols. Look back involves the ability of a logical process to change the simulation past locally without invoking change in other LPs. It was discovered when investigating ways of reducing the need to rollback in optimistic algorithms. It has been found that look back can completely eliminate the need for rollback by relying on a relaxed causality constraint made possible by LPs being able to aggressively repair any damage.

The idea of impact time is introduced to identify the maximum amount of look back that is possible and can be shown to always be larger or equal to the available look ahead and have been proven as both theoretically and experimentally faster than look ahead based conservative algorithms [46].

2.4.4.2 Composite Synchronization

Composite synchronisation combines conservative and optimistic mechanisms in an attempt to overcome the limits of using one alone. One approach developed by Hassan [47] hierarchically combines Time Warp with a conservative time window aimed at reducing cascading rollbacks and scalability to very large systems.

Nicol and Liu [48] present a composite synchronization method which combines the CCT algorithm [1] with the concept of synchronous channels which allow a more flexible timeline to be created.
2.4.4.3 **Aggressive No Risk (ANR) Algorithms**

Another major class of algorithms are ANR algorithms [49] [50]. ANR algorithms execute optimistically in a way similar to Time Warp algorithms, but do not send out new events generated by optimistic processing until they are sure no causality error will result. This removes the possibility of cascading rollbacks and the need for state saving, but limits the amount of parallelism exploited.

ANR algorithms seem more suited to simulations where each event requires some significant amount of processing since with more frequent events the likelihood of causality errors in the general case increases.

2.5 **Can the Field Survive?**

In 1993 four papers were published which called into question the survival of PDES research.

Fujimoto [13] states that the use of parallel computers to speed up discrete event simulation has not had a significant impact on the simulation community. To rectify this, the effort and expertise required by simulation practitioners to develop models must be reduced. The consequences of not achieving this will limit the effectiveness of discrete event simulation for the simulation of large-scale systems as a whole. Fujimoto identifies a set of approaches for making PDES an effective tool, which Reynolds strongly disagrees with in his reply to the article [51].

Reynolds takes the approach that “alleviating concerns about performance” is surely the silver bullet for PDES if one exists, and that once simulation builders can develop a parallel system with automated support for difficult tasks such as synchronization and communication then performance concerns will have been eliminated.

As Reynolds suggests the disagreement between himself and Fujimoto is not about what needs to be achieved, but in the approach to achieving it. For each of Fujimoto’s approaches Reynolds discusses them in the context of being distracting from the real issues and non-fundamental:

- **Application specific libraries**: many application domains have well defined primitive components; if the same could be done for PDES then model building could be simplified. Reynolds says this is has been tried for general programming but has not seen success outside of tightly defined application areas and is something which PDES cannot hold out for.

- **New languages for PDES**: much work was done in this area for sequential simulation, why not also for parallel simulation? Reynolds attacks the likelihood of a portable, special-purpose language being developed for PDES since this has not been achieved for general parallel programming research.
A Simulation Kernel

- Support for shared state; space-time memory used in conjunction with a Time Warp like rollback mechanism [52]. Reynolds says attempts using these approaches have met with limited success and experience the common performance result in parallel computation: it works well for particular problems using particular architectures.

- Automatic parallelisation; using compilers to automatically parallelise programmes. Reynolds states that many key proponents of automatic parallelisation have all but admitted that this cannot be done for the general case.

Abrams wrote an article focussing on the subject of PDES from the more pragmatic viewpoint of a simulation modeller [53], noting that programming a PDES algorithm and developing a simulation model are very different things, but are not independent, and suggests combining expertise in both areas to form a team for model development. Abrams also suggests that some kind of freely available PDES toolkit could spur interest in PDES amongst the general developer community.

Abrams mentions that Fujimoto’s approaches assume that PDES will achieve widespread use through the efforts of PDES researchers and developers, and overlook the following possible alternatives:

- Modellers with large problems to be solved and who are sophisticated about parallel architectures give up on PDES and develop their own solutions

- PDES researchers team up with simulation users to solve large, non-trivial problems. The experience of these developments feed back into the PDES knowledge base as a whole.

- Simulation modelling methodology improves so that modellers understand the technicalities of PDES programming and improve on the ways in which models are built – a kind of Parallel Modelling Methodology.

Lin [54] provides an extension to some of Fujimoto’s points, in particular that of creating new languages for parallel simulation, suggesting that any language to fill such a role should be Object Orientated (OO) due to the principles of OO design also applying to simulation model design; principally those of encapsulation and message passing. Lin describes the importance of information hiding within a development environment, so that the simulation engine can be abstracted out and hidden from the simulation model developer.

Lin also suggests that methods for the analysis and prediction of the inherent parallelism of a problem should be used before development to avoid wasting effort for minimal speedup, and gives critical path analysis [55] as one way in which this can be approached.

Some doubt from the early 90’s still exists, but since these papers were published many of the concerns have been addressed. The area where perhaps most progress has been made is in the development and availability of parallel simulation languages, indeed the ‘PDES toolkit’ that
Abrams suggests is available in several varieties (see section 2.8), at least one of which is object orientated.

2.6 Topic Summaries and Discussions

Many of the key researchers in the field of parallel discrete event simulation have published summaries of the field, along with discussion of areas showing potential and that require more research. These have served to document progress in the field and to bring together promising new research areas. Some, such as Fujimoto’s are perfectly readable without prior PDES knowledge, whilst some are aimed at more savvy users.

2.6.1 Misra, 1986

Misra’s 1986 paper titled ‘Distributed Discrete-Event Simulation’ [19] discusses parallel simulation as a promising approach to overcome the performance problems being encountered in sequential simulation, particularly in the area of network simulation. It presents a detailed explanation of the optimistic algorithm proposed in [3] including different mechanisms for deadlock detection and recovery.

Misra concludes with the following paragraph, which seems to focus on low level issues:

“[...] the most important problem in distributed simulation is the empirical investigation of various heuristics on a wide variety of problems to establish (1) which heuristics work well for which problems [...] (2) how to partition the physical system [...] (3) how to set simulation parameters such as time-outs and buffer sizes.” [19]

2.6.2 Fujimoto, 1990

Fujimoto’s invited paper for the ACM special issue on simulation [5] covers a broad array of topics and proposes future direction for research. The paper provides an excellent introduction to the problem of PDES and is one of the most cited papers in PDES literature. For conservative algorithms such as the CMB algorithm Fujimoto mentions the need to exploit look ahead, conditional knowledge and network topologies to improve performance. For optimistic algorithms such as Time Warp he points towards research in lazy cancellation, lazy re-evaluation, optimistic time windows, wolf calls and direct cancellation as areas which show promise.

Fujimoto examines the performance of both classes of algorithms and critiques their merits and pitfalls, also mentioning that the important area of simulation for real-time systems requires more work than is currently being done.
2.6.3 **Nicol & Fujimoto 1994**

Nicol and Fujimoto [36] present a selection of recent development in the area of PDES research under the headings of new protocols, analytic performance analysis, time parallelism, hardware support, load balancing, and memory management. The authors note that parallel simulation research is developing rapidly and growing in many directions and that they have tried to include relevant and important directions for future research.

### 2.7 Problems Encountered

#### 2.7.1 Relation to General Parallel Programming Problems

It is apparent that many of the problems in parallel simulation are also present in parallel processing and programming in general. Once these more general problems are solved it could be assumed that the problems in PDES would also be solved. [13] suggests this might not be the case; simulations are easier because they have relatively concrete conceptual models for starting points, and more difficult due to their dynamic, irregular and data-dependent nature. Simulation could prove to be a different nature of problem altogether.

#### 2.7.2 General vs. Specific Simulations

Throughout the methods presented so far, many are optimised for, or only applicable to, a subset of simulation problems and require explicit knowledge of these models to be implemented - one of the most difficult problems in PDES is developing general optimizations. It is demonstrated in [40] that tuning an algorithm for the purposes of running a specific simulation model can give significant speedups; this must come at the cost of generality.

#### 2.7.3 Simulation Start-up & Result Variability [56]

In 1962 Conway proposed two tactical problems in computer simulation. The first was that in a simulation model there is always some initial phase in which artificiality is present due to abrupt beginning of operation. During this phase the simulation is not representative of the running system and as such the performance and results of the simulation are unlikely to be accurate.

Conway suggests a simple solution to this problem; to exclude the data from some initial period of the simulation from the results. This seems to make sense but, as Conway also points out, the time needed before a satisfactory normal running state is achieved is difficult to estimate and also varies based on the model being run. This would seem to still be a problem today, and it falls on the model designer to provide adequate starting conditions for a simulation.
The second problem presented by Conway was that of result variability. Simulation models must often include chance and probability into their workings and as such it is necessary to estimate the precision of the results obtained.

Both of these problems are still relevant today, and have been picked up on in the model engineering community as important considerations during the model building process. As a result of this much theoretical and practical advice is available [57] [58].

Conway concludes by stating computer simulation is very much an art and that information on all aspects of simulation should be actively shared among the simulation community to prevent repeating mistakes and to aid in the progress of the field.

### 2.7.4 Causality Errors in PDES

Causality errors are the central reason PDES is so difficult [5]. Running a simulation requires the events to be processed in order – an event in the future cannot be allowed to affect something which happened in the past – but this is exactly what can happen in parallel simulation. Errors of this nature are called causality errors and they cannot be allowed to influence the results of a simulation.

### 2.7.5 Load Balancing

Balancing load across the processors available in a parallel simulation is important; ideally all processors should be fully loaded throughout the simulation. Load balancing is essential in order to effectively scale parallel simulations [59].

A naive approach to load balancing is to use a central list to store work which requires processing. As each processor completes a unit of work it fetches a new item from the list. This approach does not scale well as it introduces very poor cache locality, and the event list quickly becomes a bottleneck. Scalable performance has been seen in the CCT [1] algorithm which uses a multi-levelled approach to scheduling. A distributed memory approach to multi-level scheduling is given in [60].

A popular tactic for load balancing is to allocate LPs to processors. This introduces additional overhead if done dynamically and has proved difficult to do via static allocation due to the often dynamic distribution of workload across LPs.

### 2.7.6 Buffer Management and Cache Locality

The topics of buffer management and cache locality, and their effect on the performance of shared-memory message passing PDES algorithms are discussed in [61]. The paper examines several mechanisms for buffer management and shows that the performance of the message passing mechanism is dependent on efficient buffer management.
“Efficient buffer management is essential to achieving an efficient message passing mechanism.” [61]

2.7.7 Creating a Simulation Model which Parallelizes

In [62] Bagrodia identifies some of the common pitfalls encountered when creating models for parallel execution and provides some suggestions as to how they can be avoided. Parallelism cannot be added as an afterthought to a sequentially designed model – it requires consideration throughout the design and implementation of a model.

Some of the points identified by Bagrodia include:

- Use of shared variables and pointers. This is generally not possible in a PDES model and they cannot be encapsulated within a process and accessed using read/write messages as this is likely to become a bottleneck.

- Zero Delay Cycles. Passing a message through a process without incrementing the timestamp can lead to deadlock in most PDES algorithms.

- High connectivity between components. This will increase the synchronization overhead when converted to parallel.

- Load imbalance. The computation performed by each of the processes should be kept similar.

2.8 Computer Simulation Languages

Many simulation based languages have been created to help solve problems in building computer simulations. In 1965 Tocher [63] reviewed a selection of early simulation languages in an effort to assess the mechanisms they employed. He found that languages could be categorised by the way in which they deal with time advancement and organisation and the structuring of simulation entities within the language. Tocher found that in nearly all languages active entities such as machines, people and processes were dominant over passive entities such as materials, information and resources. A history of early simulation languages (pre 1993) is given in [64].

Tocher presented his findings in order to help simulation programmers select a language suited to their task and since this study, others [10, 65] have investigated the evolving selections of languages available. [10] especially puts emphasis on the implication of language choice with respect to developers, language implementers and language designers. With the availability of free, general purpose simulation APIs [66-70] for many popular languages the choice of language may not be as important as it once was.
I have included discussion of two languages in particular, due either to the advancements in simulation languages they provided, or the problems they aimed to solve.

The simulation kernel proposed in [66] aimed to be all of what its authors identified as the key properties of a general purpose discrete event simulator: modular, extensible, distributed, and portable. In doing this it differentiates itself from many other parallel simulation algorithms proposed.

2.8.1 MEX

The first of these languages, MEX [71], was released in 1989 and was an extension to MODULA-2. MEX is significant as it introduced primitive types to support discrete event simulation entities. The concept of a process forms the core of MEX and support for inter-process communication and synchronisation is provided. O’Dowd [71] claimed that unlike many other simulation languages MODULA-2, and hence MEX, provided good support for software engineering concepts such as types and modularity.

2.8.2 MAISIE

The second language, MAISIE [72], is noted because of its relevance to other issues discussed in this report. MASIE was designed to clearly separate the simulation model from the underlying simulation algorithm. A MAISIE program may be executed using a sequential, conservative parallel or optimistic parallel algorithm and includes optimisations within the model for each. MAISIE also provides the user with guidance for improvement within the model based on the algorithm being used. In this way MAISIE makes steps toward the second ‘silver bullet’ highlighted by Fujimoto in [13] as a way of reducing the effort needed to develop parallel models.

2.9 Application Areas

The applications to which simulation can be put to use are as varied as the systems which can be simulated and include logic circuits [73], transport networks [74], Unmanned Aerial Vehicle (UAV) research [75], jet engines [76] and storage systems [77].

One popular use within the research community is in modelling computer networks [78-82]. Computer networks lend themselves well to performance analysis for a variety of reasons:

- A network topology can be cleanly decomposed into logical processes; each node, router, and switch can be modelled as an entity in the system.

- Networks themselves are concerned with communication. Typically very little processing is required per event and this has the effect of stress testing the communication architecture within a simulation kernel.
Networks are a real-life example of the size and complexity of system for which demand for simulation is being placed.

In [83] a variety of problems encountered in large-scale network simulation are described, and new techniques to address them are proposed. One such problem is the amount of memory needed to run network simulations with large numbers of nodes; a solution is proposed which reduces the amount of state necessary in such models. The size of networking being modelled continues to increase and in this paper a million-node network is simulated using less than 1.4 GB of RAM.

2.10 CONCLUSION

Computer simulation is a field as old as computing itself, with a diverse range of applications and approaches. Traditional sequential simulation has been unable to keep pace with the demands of simulation modellers and a new range of approaches have been developed to exploit multi processor architectures.

Two main approaches have been discussed; conservative or CMB based approaches (among which the CCT algorithm is placed) for which a lower-bound on possible simulation time has been found, although not yet achieved for the general case; and optimistic approaches stemming from Time Warp, which continue to show potential for improvement. A variety of techniques and optimizations for both approaches, as well as some promising alternative approaches have been described.

A period of doubt in the field, around 1993, caused a number of leading researchers to publish papers expressing their concerns and recommendations for future research in the field, which have been summarised. A number of common and recurring problems in PDES have also been discussed and, where available, potential solutions have been mentioned.

Languages have been given context in the history of computer simulation and two standout languages have been discussed in more detail. A number of effective and freely available APIs and toolkits are now available; these have mostly addressed previous concerns about making PDES accessible and achievable for simulation modellers.
3 THE CCT ALGORITHM

This section explains the CCT algorithm. Parts are adapted from [1] and the reader is directed to this for a more in depth description, as well as formal proofs of the algorithms correctness.

3.1 INTRODUCTION

The CCT algorithm builds on the early work of Chandy, Misra and Bryant [14, 15], by extending the conservative approaches they developed with additional rules governing when an LP should be scheduled. CCT was designed with running network simulations in mind, and many of the optimizations are best realised in low event granularity models. A three-tiered scheduling mechanism is employed to allocate large grains of computation from a central queue, and critical channels are introduced as a means to identify and prioritize execution of LPs.

3.1.1 MOTIVATION

The motivation for the development of CCT was based on the limited absolute speedup seen in existing conservative (such as WaiKit), and optimistic (such as WarpKit) kernels. This limit was caused by the high system overheads associated with executing each event, and, in the case of WaiKit, the requirement of careful LP partitioning (which becomes increasingly difficult in large simulations) and its criticality to performance.

Based on where these systems do well, and where they do poorly, a set of conclusions were drawn up by Xiao et al regarding what would be required from a system to get consistently good performance. These include:

- Low per-event system overhead
- Dynamic load balancing
- Good program cache locality.

3.1.2 APPROACH TO SCHEDULING

On a shared memory multi-processor machine the traditional centralised event queue approach to simulation does not work well for two reasons. Firstly, contention over the central queue is too high. Secondly, cache locality is poor as each event is executed by a random processor, resulting in LP and event caches being frequently moved. The usual strategy to improve on this second point is to pre-allocate LPs to processors, a notoriously difficult thing to do.

Xiao et al realised that a centralised event queue may be a feasible approach if the granularity of the work obtained at each request could be increased. Additionally, if it were possible that each piece of work obtained required accessing the same LP and event buffer caches, then overall
cache locality would be improved. This idea led to the development of grouping logically related LPs into tasks. Before the scheduling mechanism can be discussed further it is necessary to introduce channels more formally.

### 3.1.3 Channels in CCT

A channel is a one-way pipe along which event messages can pass between logical processes, and a channel from LP, to LP, is denoted channel, . Channels are the only way in which event messages can pass from one LP to another, since shared variables are not permitted. Messages on a channel must be sent in strictly non-decreasing timestamp order.

Two types of channels are defined; those which join two LPs in the same task, and those which join two LPs in different tasks. A channel connecting two LPs within a task is called an internal channel, and a channel connecting two LPs in different tasks is called an external channel. This distinction becomes important when discussing the scheduling mechanisms used in CCT.

Each channel has a clock and a delay. The clock value of channel is denoted and represents the minimum timestamp value that any event message planned into the channel can have. The delay value, denoted , represents the look ahead of the channel (look ahead is explained in detail in [5]).

In addition to the clock and delay fields, each channel has three flags:

1. critical, flag to indicate if channel is critical
2. sampled, flag to indicate if there a message from channel in queue
3. busy, flag to indicate if LP is currently being executed.

The channel clock is only updated by the source LP, and the three flags are only updated by the destination LP. The source and destination LPs both access the channel event queue indirectly when sending and receiving event messages.

### 3.1.4 Logical Processes & Logical Process Scheduling

LPs are the building blocks of a simulation model and encapsulate some logical part of the system being modelled. A simulation contains a set of LPs (, ..., ), which can only communicate via time stamped event messages.

Each LP has an internal event queue, and a clock. The clock denotes the current simulation time the LP has progressed to. The event queue contains event messages ordered by timestamp belonging to that LP. Events from the queue are processed by the LP, how they are processed depends on the LP and the contents of the message.

How an LP is scheduled depends on its type, of which there are two; source LPs and non-source LPs. A source LP is one with no input channels, meaning that in the normal operation of the
algorithm it would never be scheduled. Such an LP might continue running until the simulation end time, to prevent this source LPs are allowed to run only for a fixed amount of time. When a source LP finishes an execution session, it automatically reschedules itself with its parent task, the task of which the LP is a member.

In a CCT system, each LP is in one of three logical states:

1. **Ready**; the LP has been scheduled and is waiting to be executed
2. **Executing**; the LP is currently being executed
3. **Waiting**; the LP is not currently ready or executing, and must be scheduled. This is the default starting state for all LPs.

### 3.1.5 Events & Event Scheduling

Events are the unit of communication in CCT, and are used by LPs to communicate with one another. An event has a time stamp value, representing the time it is scheduled for execution by a destination LP.

Events are scheduled for execution at their destination LP, and events are only executed when this LP is executed. The CCT algorithm aims to execute as many events as possible during each execution session of an LP, in an effort to lower the per-event overhead.

CCT calculates a safe time, or time stamp upper bound, to which an event can safely be processed without risking causality errors. This safe time is refined downwards during execution in a last fashion and based on the arrival of new information.

### 3.1.6 Tasks & Task Scheduling

A task is a group of LPs which are scheduled as a single item. The LPs which form a task should have a high degree of dependence on one another as they will all be executed on the same processor. LPs with a high dependence are identified as being likely to send event messages to each other during an execution session.

Tasks are scheduled at the highest level from a centralised task queue, and require the only system level lock in the CCT algorithm. Tasks are placed in the queue as part of the LP scheduling mechanism and in two circumstances:

1. A source LP finishes an execution phase and reschedules itself.
2. An LP from outside the task schedules an LP inside the task via an external input channel.
3.2 The CCT Algorithm

3.2.1 Constraints

There are three constraints on the allowed model behaviour, and LP topologies. Firstly, if an event with timestamp $A$ is generated as a result of executing an event with time stamp $B$, then it is required that $A \geq B$. Secondly, zero lookahead cycles of LPs are not allowed. That is, if it is possible to trace a path from an output channel of an LP, to an input channel to the same LP, the sum of the delays of those channels must be greater than zero. Thirdly, as previously mentioned, events must be placed in strictly non-decreasing timestamp order in a channel.

3.2.2 Local Window Estimation

Whenever an executing LP encounters an empty input channel, a new local window time is calculated. The local window time is updated to the empty channel’s channel time only if that channel time is smaller than the current local window time. Where channel time is defined as:

$$\max(T_{ij}, t_k)$$

Where $T_{ij}$ is the channel clock, and $t_k$ is the time stamp of the last event sent on channel $ij$. Whenever the local window estimation calculates a new local window time, the channel from which the value was derived is stored.

3.2.3 Stage 1: Initial Local Window Calculation

Each input channel is accessed, its busy bit set, and its critical bit unset. If the sampled bit is set then no further action is performed on this channel. If the sampled bit is not set, then an attempt is made to remove an event from the input channel and place it in the LPs event queue. If no message is found then the local window estimation is performed on that channel.

3.2.4 Stage 2: Event Execution

While there are events in the event queue, with time stamps less than the current local window time, the following actions are performed.

If the timestamp of the event at the head of the event queue is less than the local window time, the event is removed from the queue and executed. Next, the channel from which the executed event was received on is polled for a new event. If an event is found, it is removed from the channel and placed in the LP’s event queue. If an event is not found then the local window estimation is performed on the channel.
3.2.5 **Stage 3: Update Clock and Set Critical Channel**

The LP’s clock is set to the value of the current local window time. If the LP is not a source LP, and the LP’s clock has not advanced past the simulation end time, then the critical bit is set on the channel which currently defines the local window time. The busy bit for each input channel is unset.

3.2.6 **Stage 4: Update Channel Clocks and Schedule LPs**

Access each output channel and update the channel clock based on:

\[ \max(t_k, T_{i,j.old}, T_i + D_{i,j}) \]

Where \( T_{i,j.old} \) is the current channel clock, \( t_k \) is the timestamp value of the last event to be sent on the channel, and \( T_i + D_{i,j} \) is the value of the LP’s clock + the delay value of the output channel.

Finally, if the output channel has its busy bit set, wait for it to become unset, and then, if the channel is flagged as critical, schedule the destination LP.
4 Parallel Simulation Analysis

This chapter serves as a more in-depth introduction to relevant topics in parallel computing and parallel simulation. An understanding of these topics is important in order to make informed decisions during the development process.

4.1 Parallel Architectures

Flynn’s Taxonomy [84] consists of four computer architectures characterised by the cardinality of the instruction streams and data streams they use. These are:

- Single Instruction, Single Data (SISD): a serial, and by far the most prevalent architecture. Most desktop PCs are of this type.
- Single Instruction, Multiple Data (SIMD): a parallel architecture which executes the same instruction upon multiple data streams. This architecture is used in graphics processing and other areas where tasks can be naturally parallelised.
- Multiple Instruction, Single Data (MISD): a parallel architecture which executes different instructions on a single data stream. This architecture has had very infrequent usage.
- Multiple Instruction, Multiple Data (MIMD): a parallel architecture in which multiple instruction streams are executed against multiple data streams. This is by far the most common form of parallel architecture and is used in most modern supercomputers, grid computing and newer multi-core PCs.

Simulation lends itself naturally to the MIMD parallel architecture because each LP executes heterogeneous operations on local data - effectively ruling out the feasibility of SIMD and MISD architectures.

4.2 Parallel Overhead

Parallel overhead refers to work necessary in order to coordinate the execution of parallel tasks. Such work should be minimised wherever possible, and may include the following factors:

- Start-up, initialisation & termination; this is typically a separate task from the main execution phase and as such may not be parallelised.
- Process synchronisation; communication between parallel processes required to ensure correctness of execution. In the case of PDES this typically includes the LP scheduling mechanism.
Data communication; data required by multiple processes may cause overheads due to transfer speed (in the case of a distributed parallel system) or cache locality and access contention.

I shall, wherever reasonably possible, aim to mitigate or minimize these overheads. The CCT algorithm, and PDES in general, intrinsically requires a significant amount of synchronisation. As such, the focus of mitigation will be on the programming constructs, data types and library classes used.

4.3 Amdahl’s Law

Amdahl’s law related to parallel computing describes the relationship between expected speedup of a parallelised program, relative to the serial program. It requires knowing the proportion of a program $P$ that can benefit from parallelization (made parallel), where $(1 - P)$ is the proportion which cannot benefit from parallelization. The maximum possible speedup using $N$ processors is

$$\frac{1}{(1 - P) + \frac{P}{N}}$$

*Figure 4.1 Amdahl’s Law for Parallel Speedup*

In practise this means that as $N$ increases the speedup achieved is subject to diminishing returns. Even a small value for $(1 - P)$ has a large effect on this since it effectively rules out a proportion of the whole problem from being improved. The equation in Figure 4.1 also assumes that a program parallelizes perfectly (that there are no overheads incurred when executing a program in parallel) which is not normally possible.

Within PDES the amount of the program which is parallelizable depends on the setup and tear down time required, relative to the time required to run the actual simulation. The next section discusses factors which affect the overhead incurred when parallelizing simulations.

Figure 4.2 shows how the amount of a program that is parallelizable has a dramatic effect on the overall increase in performance achieved through parallelization. The larger the part of work which may be parallelized, the more effective any parallelization will be.
Figure 4.2 A program comprising of two tasks; Task A which takes approximately 75% of the execution time, and Task B which takes the remaining 25%. It is clear that executing Task A twice as fast gives a substantially better result than executing Task B five times as fast.

4.4 PDES Performance Factors

PDES has particular factors, both in the algorithm used and the model run, which affect the performance of a simulation. A discussion of these factors in relation to the CCT algorithm and proposed development are given below.

4.4.1 Scheduling Mechanism

The scheduling mechanism used can have a large effect on simulation performance. We have already discussed how the central event queue scheduling mechanism used in sequential simulations quickly becomes a bottleneck when multiple processors are used to execute the events.

Various scheduling mechanisms have been proposed for use in PDES, most of which employ a distributed approach. The CCT algorithm mains the central queue approach, but overcomes the bottleneck by introducing a tiered queue system.

4.4.2 Cache Locality

The principle of cache locality [85], or locality of reference as it is also known, was born during the development of virtual memory by a need to improve performance and robustness. Locality of reference relates to the phenomena that memory accessed recently will be faster to access again. At a lower level this is due to recently accessed data being stored in a faster cache, among other factors.

This principle relates to parallel architectures as each processor has a set of very fast local caches. If an LP is executing on a processor the cache for that processor will likely have cached data about that LP, increasing the speed at which the LP is executed. This is significant when we consider that it took time for the cache to be filled, and that the next LP to run may push the previous data out of these caches. In this case we would want to run an LP for as long as possible to maximise performance by minimising cache misses swapping.
The CCT algorithm provides good cache locality in a few ways:

1. Each LP is run for as long as possible, and is only run when that LP is critical to simulation advancement. This allows as many event messages as possible to be grouped at an LP before an LP is executed.

2. LPs are grouped into tasks, which are executed on a single processor. During the execution of a task any given LP is likely to be executed multiple times. The LP will run on the same processor each time, meaning that memory used by the LP is more likely to already be cached on that processor.

3. Event messages sent between LPs in a task may trigger execution of the destination LP. In this case the event message memory has temporal locality, and is likely to be in a quicker cache.

4.4.3 LP Partitioning

The manner in which LPs are chosen can affect the performance of a simulation model. Since LPs are usually the unit of execution per processor, they need to be chosen so that there are sufficient LPs to keep all available processors busy. It is also important to choose LPs to minimize the need for event messages to be sent between LPs; high communication areas of a model should be contained in as few LPs as possible.

4.4.4 Event Granularity

Event granularity refers to the computation required when processing an event message. Typically this will depend on the system being modelled, as well as the event type. In a network model, processing a message may be as simple as looking at its destination and forwarding it to the next node. In contrast a single event may require an LP to encode a video file taking minutes, hours, or days.

Each event in a PDES system introduces processing overhead in a number of ways:

- Larger Queues; more events means bigger queues with slower insert operations.
- More Communication; more events must be moved around the simulation.
- Start-up and shutdown; overheads within an LP for starting and finishing execution of an event.

4.4.5 Queue Management

Queues are used to store collections of objects in some order. In the simple case objects are removed from the queue in the order they are inserted, this is known as a FIFO (First In First Out) queue. Another form of queue is the priority queue [86], where ordering is based on some property of the inserted objects. This introduces complexity since objects are not simply inserted
at the back of the queue; their place in the queue must be determined relative to the other objects. Whilst FIFO queue operations take $O(1)$ time, priority queue operations typically take $O(\log n)$ time, although [87] presents a $O(\log \log n)$ solution.

In computer simulation priority queues are often used to order objects by their timestamp value. This is the case in the traditional event list model, as well as the three-tiered scheduling mechanism used by CCT.

The CCT algorithm tries to minimise queue management overheads by keeping queues as small as possible. The most frequently modified queue in the CCT algorithm is the LP level event queue. This queue contains at most one event from every incoming channel to the LP. The task and LP queues used contain at most the number of tasks in the simulation and LPs in the task, respectively. The largest queues are likely to be the event queues in channels, since these are strictly FIFO queues the performance overhead should be minimal.

### 4.4.6 Load Balancing

Load balancing is the process of distributing work to available resources in a way which minimizes the amount of idle time each resource encounters. Ideally, in a simulation, processors should be under 100% load at all times, but in practise this is often not possible as algorithmic constraints, I/O, interrupts, blocking and memory contention serve as possible bottlenecks.

Load balancing is important in parallel computing because the overall performance is determined by the slowest task to complete. It is especially important in PDES, where constraints on execution order form critical paths through the simulation which other areas of the simulation cannot progress ahead of. Figure 4.3 shows an example situation, where one slow process causes the overall performance to suffer.

![Figure 4.3 An Example Task Execution Pattern, with Three Tasks Waiting for a Fourth to Complete.](image)

Load balancing can be approached statically or dynamically, both with advantages and disadvantages. The CCT algorithm uses dynamic load balancing based around a central task queue which processors request work from whenever they are idle. This mechanism is very simple and introduces only a small amount of overhead when requesting work and inserting new
work into the task queue. The scalability of this mechanism depends on the granularity of work assigned from the queue; larger grains are more scalable as queue access is less frequent.

4.4.6.1 Static Load Balancing

Static load balancing requires that work is pre-allocated to a task or processor. It is typically suited to problems where the amount of work is known and can be easily divided. Static allocations do not perform well where the amount of work done by any part of a system changes dynamically. In such situations a bottleneck may form around one task. Problems requiring homogenous operations across large amounts of data, where only the amount of data and not its value defines the required work, are ideally suited to static partitioning.

Static load balancing may require some initial processing to split and allocate work, but introduces very little overhead during execution. It may be possible to include some kind of congestion recognition and recovery mechanism to reallocate workload; such a mechanism will incur additional overheads and blurs the line between dynamic and static allocation strategies.

4.4.6.2 Dynamic Load Balancing

Dynamic load balancing uses some mechanism during run time to allocate work to available resources. It is typically suited to problems which generate additional workload in a non-deterministic way, or systems in which the values of data, not the quantity determine the amount of work needed. Several popular methods for dynamic load balancing include; round-robin, priority queuing, and fastest response time.

Dynamic load balancing requires decisions to be made during execution, introducing some additional overhead.

4.4.7 Logical Process Partitioning

Logical process partitioning decisions require consideration of load balancing, LP connectivity, and processing patterns and topologies which may affect performance of a simulation. Although the CCT algorithm contains a form of automatic load balancing the problem of LP allocation to tasks remains.

In CCT, a task is a group of LPs which are executed in a certain way. Two types of tasks are introduced in [1]:

1. Pipe tasks; for groups of LPs connected in a pipeline. This task type has exceptionally good cache locality and affords a type of time parallelism as multiple processors may execute within the task at the same time, following each other along the pipe.

2. Cluster tasks; for groups of low look ahead LPs. This task type has a shared event list for member LPs and is suited for more highly connected nodes than a pipe task.
A Simulation Kernel

A bad allocation of LPs to tasks may result in problems similar with those of a poor static partitioning of LPs to processors. At worst, one task in a simulation may hold up all other tasks as they wait for events from this task. In such a situation the task should be split as best as possible to give multiple processors a chance to work on the task’s member LPs.

Identification of the node/process topologies and patterns which match those for which task types exist should form the basis of LP to task allocation. It may be possible to decide this algorithmically, affording some automation where a manual allocation is not feasible (perhaps in a simulation with very many LPs).
5 REQUIREMENTS

5.1 FUNCTIONAL REQUIREMENTS

Functional requirements are statements of service the system should provide [88]. The functional requirements of the system are the following:

1. The system must use the CCT algorithm as defined formally in section 4 of [1] for executing LPs.
2. The three tiered scheduling mechanism described in section 3 of [1] must be implemented as the primary task, LP, and event scheduling mechanism.
3. The simulation kernel must be capable of running simulations for a user defined period of simulated time.
4. The simulation kernel must be capable of detecting simulation termination.
5. The simulation kernel must be capable of executing all valid simulation models.
6. The system must be capable of utilising multiple processors when available.

5.2 DESIGN CRITERIA

This section introduces the design criteria set for the implementation of the CCT algorithm and the reasons for these choices. During the design and implementation of the project these criteria will be used to guide decisions in an effort to ensure the best end product. Some practical considerations based on the author’s experience, as well as constraints such as development time are included. The criteria below are discussed in more detail in the following sections.

1. Usable
2. Efficient
3. Extensible
4. Simple
5. Portable.

5.2.1 USABLE

The kernel has to be usable for model developers and people running simulations. It should be as straightforward as possible to develop models using the kernel as a base, and should provide
help where necessary. In order to achieve this helper functions which abstract out the more technical activities of the system may be needed.

The system should also be usable from the standpoint of a person running simulations. It should be possible to automatically repeat simulation runs and collect aggregate data from those runs. The simulation runner should not need any knowledge of how the system works to be able to use it.

5.2.2 Efficient

The primary aim of PDES is to improve simulation performance; the implementation should therefore be as efficient as possible. Language constructs and data types which are slow should be avoided, and code libraries will be chosen with performance in mind. The code should only contain essential functionality to keep memory usage down.

5.2.3 Extensible

The system should be designed in a way that allows easy extension of the base kernel by another developer. It is essential that a developer can create new task types, which execute differently from built in tasks, and new event message types since this will almost certainly be needed by the model developer. Ideally it will be possible to extend all of the concepts used in CCT, and to reconfigure existing ones for tuning purposes.

5.2.4 Simple

The implementation should be kept as simple as possible in an effort to keep the system easy to maintain, understand, and debug. The design should be a loosely coupled as possible to contain the effect of extensions and modifications. The KISS (Keep it Simple, Stupid) principle should be followed wherever possible. This principle also extends to the interactions of the simulation model developer, it should be simple to develop and hook in a new model to run on the kernel.

One aim of keeping the system simple is that a user with some programming experience, and some knowledge of CCT should be able to follow the system, and see how it is implemented.

5.2.5 Portable

Portability on machine architecture gives the user greater freedom in how and on what computer simulations can be run. Portability usually involves some decrease in performance (see section 2.7.2), so this design principle conflicts with the above efficiency criteria; such conflicts will be resolved by evaluating which principle would provide the greatest benefit if followed.
5.3 Version Control Software

Version Control Software (VCS) should be used throughout the development process as a means to manage the code base. VCS ensures a safe backup of all the project code, as well as providing a historical trace of all changes made over time. Using VCS was deemed essential, to protect against loss of work due to hardware failure, and to create a properly versioned and documented change history.
6 DESIGN & IMPLEMENTATION

The CCT algorithm is clearly defined, but implementing a simulation kernel based on it requires consideration of many factors outside the scope of the algorithm. The design process is documented, starting with the high level system flow, and then breaking down the main components into more detail.

6.1 DESIGN

6.1.1 LANGUAGE CHOICE

The project will be implemented in Java, more specifically the latest version, Java 2 Platform SE 5.0. This choice was made after considering the following factors; the authors programming experience, which is mostly in Java, the availability of development tools, and the API available. As an Object Oriented (OO) language Java provides class inheritance and interfaces, which can be used to build an extensible system. Java is also portable, meaning the system can be used on multiple computer architectures and operating systems. OO languages, and Java especially, are popular at the moment, and using a language users are familiar with may help model development.

Java includes some language features which benefit large scale applications; dynamic linking[89] means that classes are not loaded into memory until required, and garbage collection [90] provides an efficient memory reclamation mechanism, reducing the burden on simulation and model developers.

Perhaps the biggest disadvantage of using Java is that it is not a fast language. It is executed via a virtual machine which translates the compiled Java program to native machine instructions (although this does mean the language provides portability). A language such as C or C++ would most likely give better performance, but the authors experience with Java, and the time required to learn a new language within the time constraints on development lead to choice of Java.

6.1.2 OBJECT ORIENTATED DESIGN PRINCIPLES

Several Object orientated design principles are identified in [91] and can be used to guide good class design. Decisions made during the design process should take into account these principles which are aimed at creating simple, loosely coupled, and easily maintainable code, which helps towards the usability and the simplicity of the end system.
6.1.2.1 ENCAPSULATION

Encapsulation is the idea of containing and hiding the contents of a part of the system from the other parts except where explicitly required. Each subsystem should require access to other subsystems as infrequently as possible, and should be designed so that as few as possible other subsystems require anything from it. Three access modifiers are used to define the strictness of the encapsulation used:

1. **Private**: only the defining class can access private members
2. **Protected**: the defining class and any subclasses can access protected members
3. **Public**: everything can access this member

The rule of thumb is to hide as much as possible as fully as possible, and a good practise is to declare everything as private initially, and relax this restriction only when required. Using *getter* and *setter* methods for class variable access is better than allowing direct access to variables and allows various pre/post processing to be added in the future without changing the classes public interface.

6.1.3 SYSTEM FLOW DIAGRAM

A high level outline of the system flow in Figure 6.1 shows the inputs and outputs of the intended system. This is perhaps a very simplistic view of the system, but it shows the simulation kernel as the central element of the system. It should be clear from the diagram that logging information is output during execution (if enabled), and allows tracking of simulation progress.

![System Flow Diagram](image)

*Figure 6.1 The basic, high level system flow. A simulation model is the input, and the simulation results, and logging information are the outputs.*

6.1.4 CLASS DESIGN

The first step in designing the system was to identify the key components which could form the core set of classes in the system. Some of these components were identified in section 3.1. The relationships between these classes were analysed, resulting in the following class diagram (see...
Figure 6.2). Note that class fields are accessed via getter and setter methods which are not included in the class diagrams for brevity.

<table>
<thead>
<tr>
<th>Task</th>
<th>Simulation</th>
</tr>
</thead>
<tbody>
<tr>
<td>-memberLPs : ArrayList&lt;LogicalProcess&gt;</td>
<td>-memberTasks : PriorityQueue&lt;Task&gt;</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Logical Process</th>
<th>Channel</th>
<th>Event</th>
</tr>
</thead>
<tbody>
<tr>
<td>-inputChannels : ArrayList&lt;Channel&gt;</td>
<td>-eventList : LinkedQueue&lt;Event&gt;</td>
<td>-time : int</td>
</tr>
<tr>
<td>-outputChannels : ArrayList&lt;Channel&gt;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-eventQueue : PriorityQueue&lt;Event&gt;</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Figure 6.2 Initial class diagram showing basic classes and fields specified in the algorithm definition.**

Section 4.1 of [1] describes the necessary state for LPs and channels, these are added to the relevant classes.

### 6.1.4.1 Algorithm Walkthrough

Based on the initial structure additional attributes and behaviours it is possible to begin stepping through the CCT algorithm, and adding functionality as it is required.

#### 6.1.4.1.1 Initialisation

The system state is initialised, and the default starting values for simulation objects are set. Since this only happens once per simulation run it is encapsulated in the class constructors and default field values.

#### 6.1.4.1.2 Local Window Time Estimation

Whenever an empty input channel is encountered the local window time for that LP must be updated. The information required for this must be accessible by the LP; the input channel clock, the timestamp of the last event to pass along the channel, and the clock value of the LP being executed.

A new method `updateLocalWindow` in the Logical Process class is required, which will be called whenever an empty input channel is encountered. The method will update the local window time, and store a reference to the input channel from which the value was calculated.

#### 6.1.4.1.3 Poll Input Channels

When an LP begins execution, the input channels are polled to determine if they are non-empty. If an event is present then it is removed and placed in the LP’s event queue. If an event is not
present then the local window time is calculated based on the current channel. A new method `pollInputChannels` will perform this task, and will be called whenever an LP begins an execution phase.

As channels are accessed the busy flag must be set, and the critical bit unset. It makes sense to combine these actions in a new method, `initialiseProcessing`, in the Channel class.

### 6.1.4.1 Event Execution

While there are events in the event queue of an LP, with timestamps less than the local window estimate, these events are removed from the queue and executed. When an event is removed, the input channel it came from is polled for another event. These actions are placed inside a new method `executeEvents` in the Logical Process class. This is perhaps the most critical loop in the LP class, and a slightly simplified code snippet is given in Appendix C.1 for further detail.

The LP clock can be advanced manually at this stage to take advantage of any additional look ahead that may be present in the simulation. A new method `advanceClock` sets the LP clock to a new value in the future.

### 6.1.4.1.5 Update Logical Process Clock

When all events up to the safe time have been processed the LP’s clock is updated to that safe time. The input channel from which this safe time was calculated is set critical, and all input channels have their busy bit unset.

These operations are all performed every time an LP finishes executing events and can be combined in a new method `finishExecutionPhase`.

### 6.1.4.1.6 Update Output Channel Clocks

At end of each execution phase the output channel clocks are updated, a new method in the Channel class `updateClock` is needed, and is called as part of the `finishExecutionPhase` method in the Logical Process class. It is also necessary to keep track of whether during a given execution phase a message is sent along a channel, a new variable `updatedDuringCurrentExecution` is introduced.

The `updateClock` method is included in the Channel class because it relies on information in the channel object which should not be made public. Since the LP’s current clock time is required as part of the calculations this is passed in as a parameter to the method.

### 6.1.4.1.7 Schedule Destination LPs

When an LP finishes an execution phase, the critical flag on the output channels for that LP are checked, and if set to true then the destination LP for the channel is scheduled. Since an LP does not store references to the LPs connected to its output channels, scheduling needs to be
performed by the output channel objects, which do reference their destination LPs. This scheduling can be incorporated into the `updateClock` method in the channel class.

An LP is scheduled differently depending on whether it is scheduled by an *internal* or *external* channel. Each channel needs to store this value, which can be determined when the channel is created. A new variable, `isExternal` in the Channel class is required. Based on the value of this variable one of two new methods in the Logical Process class, `scheduleLocally` and `scheduleGlobally`, is called to schedule the LP at the correct level.

When an LP is scheduled locally it is scheduled in the task it belongs to, requiring a new method in the Task class, `scheduleLP`. When an LP is scheduled globally it is scheduled in the task it belongs to, and the task itself is scheduled in the Simulation class. Two new methods are needed; `scheduleGlobalLP` in the Task class, and `scheduleTask` in the Simulation class. In order to call these methods a logical process needs a reference to its parent task, and a task needs a reference to its parent simulation.

Using this mechanism, a call chain is used to pass the scheduling request “up” the scheduling hierarchy until the appropriate level is reached. Such a call chain requires that each class needs only to know about the class one level higher.

### 6.1.5 Updated Class Diagram 1

The new class diagram (Figure 6.3) shows all of the methods mentioned in the algorithm walkthrough. Some new variables, such as `parentTask` and `parentSimulation` enable the call chain for scheduling LPs to be implemented.

At this stage the framework for implementing the CCT algorithm are all in place; what is not considered yet is how a simulation model can fit into this framework, and how actual execution proceeds. The next section describes how a simulation model can be implemented, and the modifications and additions necessary to the above base classes for this to take place.
A Simulation Kernel

### 6.1.6 Building Simulation Models

The simulation modeller requires additional functionality to be able to create a model which will work with the simulation kernel. These requirements are listed below, and the implementation details are discussed in the following sections.

1. A way of defining the logical processes in the simulation model
2. A way of defining event messages for communication between logical processes
3. A way to create tasks, and assign LPs to those tasks
4. A way to connect LPs via channels
5. A way to send messages from one LP to another
6. A way for an LP to schedule event messages for itself
7. A way to define and initialise the simulation model being used.

#### 6.1.6.1 Defining Logical Processes

The simulation modeller must be able to define logical processes which implement the behaviours required of the simulation model. How these logical processes are determined is outside of the scope of this project, but how the simulation modeller can define them is not.
One approach would be to define an interface which the simulation modeller must implement, and which can then be called by the base Logical Process class. When a Logical Process enters the executeEvents method, this could call the modellers class that implements the interface and pass in the events to be executed. One disadvantage to this approach is that the simulation modeller would have no access (except via explicitly programmed accessor methods) to any methods or data from the base Logical Process class. Another disadvantage is that the mechanism does not seem intuitive, every event is passed into another class.

A different approach would be to declare the Logical Process class as abstract, and to require the simulation modeller to extend the class and implement some abstract methods in their logical processes. This seems a natural use of the ‘is a’ method for identifying a class hierarchy during class design, since a logical process in the simulation model is a more specialised version of the base Logical Process class. Events being executed would still need to be passed into the modeller’s class, but performance would be improved as polymorphism is used to pass the event down the class hierarchy automatically. Additionally, the simulation modeller would have easier access to methods and data used in the base Logical Process class, and the protected visibility keyword could be used to share these with only these sub classes. Finally, such an approach would allow the simulation modeller to easily continue the LP hierarchy within their own model’s classes.

With the extensibility design principle in mind, the sub classing approach is the better option. The Logical Process class becomes abstract, and defines the abstract method processEvent(Event event). This method is called as part of the executeEvents method in the Logical Process class every time an event is removed from the queue to be processed.

6.1.6.2 Defining Event Messages

The simulation modeller must be able to define new types of event messages. These event messages are passed between logical processes to exchange information and schedule work.

There are two options as to how this is implemented; one is to use the Event class as a wrapper for some contained Object, the other is to allow the simulation modeller to subclass the Event class and implement additional functionality in this subclass.

The advantage of using the subclass approach is that it allows the modellers event classes to access methods and data in the base Event class. It is likely that the simulation modeller will need to know the event timestamp, and using inheritance this can be accessed via protected visibility methods. Using the wrapper approach also forces the simulation modeller to cast their event messages from type Object into the type they require. This inefficient operation is removed in the subclass approach by defining an abstract method, execute, in the Event class.
A Simulation Kernel

which can optionally be implemented by the modeller in a subclass. By implementing this method the model developer can encapsulate work within the event objects they define, without needing to cast the Event object, and can use their logical process classes to instead organize the work that needs to be done. It is also possible for the logical process to perform operations on the event message as required.

6.1.6.3 Task Creation and Logical Process Assignment

The simulation modeller must be able to define the tasks within their model, and assign LPs to these tasks. Tasks are always executed by a single Simulation class, and the simulation class needs to register any tasks that are added so that they can be scheduled initially when the simulation begins. Once tasks have been created they can be added to the simulation via a new method, addLogicalProcess.

Similarly, a logical process is always executed by a single task, and that task needs to keep track of the LPs which it has scheduled. A new method in the Task class, addLP, registers an LP with a task, and schedules it to be executed when the task is next executed. It is important to schedule the LP when it is added since it may not be scheduled by any part of the simulation, and the model will fail to execute.

6.1.6.4 Connecting Logical Processes via Channels

The simulation modeller must be able to define channels between the logical processes they create. This would require knowledge of the simulation kernel, as well as updating data structures on the source and destination LPs. Keeping in mind the simplicity design principle, it is possible to provide a helper method which performs these tasks in a more user friendly way. A new method, createOutputChannel, is added to the Logical Process class, taking as parameters the destination LP, and a delay value for the channel to be created. These are passed into the constructor of a new channel object which sets the channel delay, the source and destination LPs, and determines whether the channel is internal or external. The destination LP also needs to be notified, and update its list in incoming channels. This is handled by a new method, registerInputChannel, which is called by a Channel when it is created.

One disadvantage to this method is that it requires both the source and destination LPs to be assigned to a task before the channel can be created (so that the internal/external check can be made). A check of this is made at run time, and an exception is thrown with a message describing what has occurred if this is not the case. This affects the simulation modeller by creating a constraint on the order in which a model can be built. However, since all LPs must be assigned to a task at some point, it does not introduce any additional work.

A way to fix this problem would be to not require the source and destination LPs to be assigned to a task when the channel is created. Instead, when either the source or destination LP for a channel is set a check is made to compare the tasks these LPs are members of, and the
isExternal value updated based on this. Another check would be required to ensure that every LP is assigned to a parent Task before the simulation execution begins. This introduces additional complexity to the system, going against the design principle of simplicity, without providing any additional functionality and very little usability, and so is not included.

6.1.6.5 Sending Event Messages between Logical Processes

The simulation modeller requires the ability to send messages from one LP to another. During the design phase of the model the modeller will almost certainly identify logical processes that will need to communicate.

During execution there are several ways an event message could be sent; direct reference to the target LP, reference to the output channel the message is to be sent on, querying some ‘oracle’ service with search criteria about the target LP, and many more exotic mechanisms. Sending an event message is likely to be an extremely frequent process, and as such the overheads should be kept to a minimum by keeping the process simple. The simulation modeller has already defined which LPs are connected by channels, so requiring only the directly connected destination LP for a message adds no additional burden to the modeller.

A new method, scheduleExternalEvent, is added to the Logical Process class, which takes as parameters a destination LP, and an event to send. This requires that the simulation modeller store references to destination LPs within their logical process classes; not an ideal scenario but it allows the overheads for sending messages to be kept to a minimum; requiring slightly more memory, but fewer processor instructions. A HashMap called channelMap, is used to store the mapping between destination logical process and the relevant output channel.

HashMap’s allow efficient lookup and retrieval of linked data. In a HashMap a key is used to lookup a value. In this case the key is the destination logical process, and the value is the channel. A HashMap is preferred over other (key, value) mechanisms (such as a Dictionary) because the lookup speed is slightly higher, and the modification costs are not a factor since all channels will remain static after the initial allocation. A HashMap does not allow duplicate key values which, in this system, this would present a problem if multiple channels between the same LPs were created. Such a construct would provide no benefit to the system and therefore this constraint does not pose an issue.

Once an event message is sent by a logical process, the event is added to the event list in the output channel. At this point if the event timestamp is less than that of the previous message sent on the channel an exception is thrown; such a message cannot be sent without violating a constraint of the CCT algorithm (see section 4, bullet 2 of [1]). A new method, sendEvent, in the Channel class performs this check, and adds the event to the channel’s event list.

This mechanism could be extended in the future to include a routing mechanism between LPs and tasks which forwards messages between any LPs which have a path of channels between
them. Such an extension would require considerable effort, and introduce much additional complexity whilst improving the usability of the system from the simulation modeller’s perspective significantly. Such a routing mechanism may however lead to poor model design, by removing incentive for the modeller to thoughtfully plan the LP and channel topologies and instead relying on the routing mechanism to deliver event messages and introducing much additional overhead. A routing mechanism such as this would be better implemented at a higher level, as some system which runs on top of the LP/channel mechanism.

**6.1.6.6 Scheduling Self Events**

The simulation modeller must have some mechanism for allowing an LP to schedule additional work for itself. This is achieved by allowing an LP to place event message in its own event queue. This could be implemented either by defining a new method, or by adjusting the existing scheduleExternalEvent method to check if the destination is the same as the current LP.

Since performance is of primary concern in such a high frequency function, adding an additional check within the scheduleExternalEvent method is unwanted. Additionally, sending an event message, and scheduling more work for the current LP are quite different operations even though they are both achieved via the use of Event objects. The two operations are separated, and a new method, scheduleSelfEvent, is added to the Logical Process class.

**6.1.6.7 Initialising the Simulation Model**

There must be some mechanism for initialising the simulation model before execution of the model begins. During initialisation the logical processes, channels, tasks and initial events can all be defined and created by the simulation modeller. Essentially this is where the components of the model are wired together ready for the simulation to begin. The Simulation class represents the highest level of abstraction in the system, and provides a convenient place for this.

There are two ways in which this can be approached; either by creating a subclass of the Simulation class and defining an abstract method, to be implemented by the model developer, or by defining an interface which the model developer implements, and is executed before the simulation is run. If an interface is used, then a reference to the Simulation object must be passed into the implementing class because the Tasks need to be registered. Such a mechanism seems less elegant than for the simulation modeller to create a subclass of Simulation, which has the addTask method available through inheritance. A new abstract method, buildSimulation, is added to the Simulation class.

**6.1.7 Process for Building A Model**

The process for developing a model to be used by the simulation kernel is now as follows:
1. Define logical processes which extend the Logical Process class, and implement the abstract `processEvent` method. The Logical Process class can form the root of an LP hierarchy if required.

2. Define event messages which extend the Event class. Optionally implement the `execute` method in the Event subclass to encapsulate work within the new event.

3. Define a subclass of Simulation, and implement the `buildSimulation` method. In this method the necessary tasks are created, and the logical process subclasses defined in step 1 can be instantiated and assigned to tasks.

4. Assign tasks to the simulation using the `addTask` method.

5. Channels are defined between the logical process subclasses defined in step 1.

6. Initial events can be scheduled at a logical process using the `scheduleSelfEvent` method.
### 6.1.8 Updated Class Diagram 2

<table>
<thead>
<tr>
<th>Task</th>
<th>Simulation</th>
</tr>
</thead>
<tbody>
<tr>
<td>- memberLPs : ArrayList&lt;LogicalProcess&gt;</td>
<td>- memberTasks : ArrayList&lt;Task&gt;</td>
</tr>
<tr>
<td>- parentSimulation : Simulation</td>
<td>- taskQueue : PriorityBlockingQueue&lt;Task&gt;</td>
</tr>
<tr>
<td>+ scheduleLP(in logicalProcess : Logical Process)</td>
<td>+ scheduleTask(in task : Task)</td>
</tr>
<tr>
<td>+ addLP(in logicalProcess : Logical Process)</td>
<td>+ buildSimulation()</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Logical Process</th>
<th>Event</th>
</tr>
</thead>
<tbody>
<tr>
<td>- inputChannels : ArrayList&lt;Channel&gt;</td>
<td>- time : int</td>
</tr>
<tr>
<td>- outputChannels : ArrayList&lt;Channel&gt;</td>
<td>+ execute()</td>
</tr>
<tr>
<td>- eventQueue : PriorityQueue&lt;Event&gt;</td>
<td>+ Event(in time : int)</td>
</tr>
<tr>
<td>- clock : int</td>
<td></td>
</tr>
<tr>
<td>- localWindow : int</td>
<td></td>
</tr>
<tr>
<td>- localWindowChannel : Channel</td>
<td></td>
</tr>
<tr>
<td>- channelMap : HashMap&lt;LogicalProcess, Channel&gt;</td>
<td></td>
</tr>
<tr>
<td>- parentTask : Task</td>
<td></td>
</tr>
<tr>
<td>+ updateLocalWindow(in inputChannel : Channel)</td>
<td></td>
</tr>
<tr>
<td>+ initialiseProcessing()</td>
<td></td>
</tr>
<tr>
<td>+ pollInputChannels()</td>
<td></td>
</tr>
<tr>
<td>+ executeEvents()</td>
<td></td>
</tr>
<tr>
<td>+ finishExecutionPhase()</td>
<td></td>
</tr>
<tr>
<td>+ scheduleLocally()</td>
<td></td>
</tr>
<tr>
<td>+ scheduleGlobally()</td>
<td></td>
</tr>
<tr>
<td>+ executeEvent(in event : Event)</td>
<td></td>
</tr>
<tr>
<td>+ createOutputChannel(in destinationLP : Logical Process, in delay : int)</td>
<td></td>
</tr>
<tr>
<td>+ registerInputChannel(in incomingChannel : Channel)</td>
<td></td>
</tr>
<tr>
<td>+ scheduleExternalEvent(in event : Event)</td>
<td></td>
</tr>
<tr>
<td>+ scheduleSelfEvent(in event : Event)</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Channel</th>
</tr>
</thead>
<tbody>
<tr>
<td>- eventList : LinkedQueue&lt;Event&gt;</td>
</tr>
<tr>
<td>- destinationLP : Logical Process</td>
</tr>
<tr>
<td>- sourceLP : Logical Process</td>
</tr>
<tr>
<td>- clock : int</td>
</tr>
<tr>
<td>- delay : int</td>
</tr>
<tr>
<td>- isCritical : bool</td>
</tr>
<tr>
<td>- isBusy : bool</td>
</tr>
<tr>
<td>- isSampled : bool</td>
</tr>
<tr>
<td>- isExternal : bool</td>
</tr>
<tr>
<td>- updatedDuringCurrentExecution : bool</td>
</tr>
<tr>
<td>+ Channel(in sourceLP : Logical Process, in destinationLP : Logical Process, in delay : int)</td>
</tr>
<tr>
<td>+ updateClock(in sourceLP:clock : int)</td>
</tr>
<tr>
<td>+ initialiseProcessing()</td>
</tr>
<tr>
<td>+ terminateProcessing()</td>
</tr>
<tr>
<td>+ sendEvent(in event : Event)</td>
</tr>
<tr>
<td>+ getNextEvent()</td>
</tr>
</tbody>
</table>

**Figure 6.4** Class diagram showing base algorithm and simulation model construction methods and data structures.
The system now supports the algorithm mechanics and allows simulation models to be built, some helper methods have been designed to make the system easier to use and compromises have been made to ensure that performance does not suffer.

6.1.9 EXECUTION AND SCHEDULING

In this section the execution strategy and scheduling details are discussed. This is essentially the core of the system and discusses how the threads of execution are managed. There are several aspects of this which will be discussed separately; task scheduling, LP scheduling, event scheduling, thread management, and termination detection.

Some of the API classes used are quite complex and only key features are mentioned in this document. A reference to the full specification of each class used is included for more detail.

6.1.9.1 TASK SCHEDULING

Tasks are scheduled at the Simulation class level, and are stored in a task queue. As execution proceeds, tasks are removed from the head of the queue and assigned to an available processor. In the description of CCT the ordering of tasks is described as “in increasing time order”, this has been taken to mean the value of the least progressed LP within the task (such a thing would be trivial to change).

Several queuing data structures are available in the Java API that could be used for the task queue, a PriorityBlockingQueue [92] was chosen for the following reasons:

1. It is guaranteed thread safe
   a. This is required since it is possible that multiple threads will be accessing the queue at the same time
   b. It is possible to synchronise access manually to the queue, but an internal implementation is more efficient, and is recommend by the Java API.

2. It implements the Queue [93] interface, meaning it can be used with a ThreadPoolExecutor (see section 6.1.9.4 for more detail).

3. It allows sorting using Comparable [94] objects. A new class TaskComparator is defined which is used in the task queue to order tasks based on their time
   a. This allows elements to be ordered based on their ‘natural ordering’.

4. It has no maximum size, and grows efficiently.

In order to allow time parallelism, multiple instances of a task can be present in the task queue at the same time. This is necessary because objects remain in the queue until their execution has finished. An extension of this mechanism would be necessary to avoid unnecessary thread
blocking if different task types, which did not allow time parallelism were to be included in the system.

The Task class implements the Runnable interface, and the run method is called to execute a Task in the task queue. The run method processes all the scheduled LPs at that task before returning.

### 6.1.9.2 Logical Process Scheduling

Logical processes are scheduled within a task, when an and are stored in a LinkedList [95] which orders elements by insertion order, and has no maximum size. The task type implemented can have multiple threads executing within it at the same time, which requires some additional work to achieve.

Firstly, each thread operating within a task maintains a separate LP queue, this is possible because every Thread [96] has a guaranteed unique thread identification number. A HashMap in the Task class is used to map each Thread Id to a corresponding LinkedList containing the LPs scheduled for that Thread. Any LPs which are scheduled by a thread as part of the tasks execution will be added to the threads private LP queue.

Secondly a ‘start-up’ LinkedList of LPs is maintained, which stores the LPs which have been scheduled from another task, and are awaiting a thread to start execution of the task and pick them up. Access to the start-up queue is synchronized so only one thread can access it at any time. When this happens the contents of the start-up queue are moved to the threads’ private queue. While this thread is executing LPs in the task it is possible for another thread to pick up a new set of start-up LPs and create another private LP queue. This allows multiple threads to execute concurrently within a thread without interference.

A task allows only one thread to execute a given LP at a time, and this is achieved using a synchronised [97] block around the actual execution of a LP. When an LP is executed the runLogicalProcess method is called, which performs the full execution cycle for that LP.

The code snippet in Appendix C.2 shows the main loop within a task which executes LPs.

### 6.1.9.3 Event Scheduling

Each LP maintains an event queue that is implemented by the PriorityQueue [98] class, which is similar to the PriorityBlockingQueue used in the Simulation class except that since only one thread will be executing a given LP at any time thread safety is not required. The event queue is ordered in increasing order of event timestamp value. A new class, EventComparator is used by the priority queue to order events. Events are removed from the queue one by one and executed by the LP.
Source LPs do not require much of the processing which is performed in non-source LPs. When a source LP is executed a separate method, `generateEvents`, is called, which must be implemented in source LPs, and is defined by a new interface, `SourceLogicalProcess`. A value, `sourceLPRunningTime`, is defined in the Logical Process class which specifies an upper bound on the timestamp value of event messages which can be generated during execution of this method. The local window of the source LP is then updated by adding this value to the current LP clock value.

### 6.1.9.4 THREAD MANAGEMENT

The simulation starts with one thread of execution entering the `Simulation` class’s `doSimulation` method. This initial thread calls the `buildSimulation` method previously mentioned, and then initialises the additional threads used during the simulation. During the simulation this initial thread also performs termination detection, described in section 6.1.9.5. When all tasks have advanced past the simulation end time simulation execution ends.

Java Thread [96] objects are used to manage multiple execution threads in the system. The `ThreadPoolExecutor` [99] class from the Java API is used to manage the processing of the task queue which is passed in as a parameter. The number of threads managed by this pool is determined beforehand either by a parameter passed in by the user, or programmatically via the `Runtime.getRuntime().availableProcessors()` method.

The number of threads in use by the simulation does not change, and the threads are recycled and not disposed of after processing a task, avoiding costly thread creation overheads. Use of the `ThreadPoolExecutor` class requires that all objects scheduled for execution implement the `Runnable` interface, which the `Task` class does.

### 6.1.9.5 SIMULATION TERMINATION

The simulation should terminate when all LPs have advanced their clocks past the simulation end time. The simulation end time is defined by a parameter passed in by the user. During execution of the simulation any event messages with timestamps greater than the simulation end time are not sent, and LPs are not rescheduled if they have passed the simulation end time. This avoids wasting time processing LPs and events which do not count towards the result of the simulation and which otherwise might affect the correctness of those results.

### 6.1.10 UPDATED CLASS DIAGRAM 3

The complete simulation kernel has now been discussed and the updated class diagrams are shown below. It is expected that during the implementation process these designs may change slightly, as practical difficulties are worked out.
**Logical Process**

- inputChannels : ArrayList<Channel>
- outputChannels : ArrayList<Channel>
- eventQueue : PriorityQueue<Event>
- clock : int
- localWindow : int
- localWindowChannel : Channel
- channelMap : HashMap<LogicalProcess, Channel>
- parentTask : Task
- isSourceLP : bool

+ updateLocalWindow(in inputChannel : Channel)
+ runLogicalProcess()
+ initialiseProcessing()
+ pollInputChannels()
+ executeEvents()
+ finishExecutionPhase()
+ scheduleLocally()
+ scheduleGlobally()
+ executeEvent(in event : Event)
+ createOutputChannel(in destinationLP : Logical Process, in delay : int)
+ registerInputChannel(in incomingChannel : Channel)
+ scheduleExternalEvent(in event : Event)
+ scheduleSelfEvent(in event : Event)

**Figure 6.5 The Logical Process abstract class diagram.**

**Channel**

- eventList : LinkedQueue<Event>
- destinationLP : Logical Process
- sourceLP : Logical Process
- clock : int
- delay : int
- isCritical : bool
- isBusy : bool
- isSampled : bool
- isExternal : bool
- updatedDuringCurrentExecution : bool

+ Channel(in sourceLP : Logical Process, in destinationLP : Logical Process, in delay : int)
+ updateClock(in sourceLPClock : int)
+ initialiseProcessing()
+ terminateProcessing()
+ sendEvent(in event : Event)
+ getNextEvent() : Event
+ sendEvent(in event : Event)

**Figure 6.6 The Channel class diagram.**
A Simulation Kernel

Figure 6.7 The Task class, which implements the Runnable interface.

Figure 6.8 The Simulation abstract class diagram.

Figure 6.9 The EventComparator and TaskComparator classes which implement the Comparator interface. The event abstract class and SourceLogicalProcess interface.
6.1.11 Remarks On The Design

6.1.11.1 Duplicated Data

The design includes some duplication of data, for example the sourceLPRunningTime and the simulation end time are stored locally in each LP. These values never change so it would be correct to allow all LPs to reference the same value, except that shared data is not possible in a distributed system and so it is necessary to duplicate some data.

6.1.11.2 Data Type of Timestamps

It seems that the general convention for representing time in simulations is based on floating point numbers, with each whole number representing one minute, or hour of real time. The design uses integers instead, purely for performance reasons. The most frequent operation in the algorithm is a timestamp compare, to decide ordering of events in LPs and LPs in tasks and so a decision was made against using floats.

This decision does affect the precision with which events can be scheduled, but by using a sufficiently large value to represent each one minute/hour time period an arbitrary precision is possible, up to the limit of the int data type. If this limit becomes a limiting factor in the model being run, the time values used throughout the system can be changed to a type of long, giving a significant increase in the maximum value.

6.2 Model Design

In order to both test the correctness of the model, and collect performance results, it is necessary to implement a set of models which utilize the full functionality of the system. Two models have been developed, which stress different parts of the system; a very simple pipe simulation for demonstrating time parallelism, and a more general purpose model built around a request-response paradigm.

6.2.1 Pipe Model

The pipe simulation model consists of a single task, containing several LPs connected in a pipe. The purpose of the model is to demonstrate the time parallelism possible in the system, and hopefully during execution the available processors will all be able to execute concurrently on the same task. The basic model can be modified to include various LP counts in the task, and at each LP some arbitrary processing is possible, allowing modelling of higher granularity events in the same topology.

The model consists of three types of logical processes:

- Sender, a source LP which generates events and sends them on
A Simulation Kernel

- Forwarder, receives events and forwards them on to the next node
- Receiver, receives events and then discards them

Figure Error! Reference source not found. shows an example pipe simulation topology, where four LPs are connected in a line by a single channel.

![Pipe Simulation LP topology](image)

Figure 6.10 An example Pipe Simulation LP topology.
6.2.2 Request Response Model

The request response model is based around the typical high-level network scenario of multiple clients making requests against some provider, with the results being forwarded back to the client. The model aims to test how scaling the size of the simulation model can affect performance, and can be generated programmatically. The model can be broken down into the following sections:

- **Clients**, the client processes generate an initial set of requests to a random server, and whenever a response is received a new request is sent out.

- **Client Groups**, clients are organized into groups, with a shared gateway onto the network.

- Each client group has an incoming gateway and an outgoing gateway. When the incoming gateway receives a response, it forwards it to the correct client. When an outgoing gateway receives a request it forwards it onto a random route towards the server. Each client group and its associated gateways form a single task.

- **Incoming and outgoing network routes** forward messages from the client gateways to the servers and back again. Each route consists of a single pipe task.

- **Servers**; each server is in a separate task. The server receives requests and performs some arbitrary processing before sending a response back to the originating client.

Figure 6.12 shows the general network topology. In order to keep each LP simple, a number of different LPs are defined, which perform specialised tasks. A more general routing algorithm for
event messages could have been implemented, but would add complexity without adding anything useful to the simulation mode.

6.2.2.1 CLIENTS

During the initialisation of the simulation, each Client schedules a number of self events. When processing begins, these self events are executed which forwards them to the outgoing gateway for that client group. These initial events are recycled throughout the simulation, so that when a response is received to a request event a new request event is created and sent out.

6.2.2.2 CLIENT GROUPS

Client groups include a number of client LPs, along with an incoming and outgoing gateway which acts as a relay for event messages entering and exiting the group. Each client group forms exactly one task. The outgoing forwarders are aware of all paths to the servers, and when forwarding each message choose a random path to forward on, creating an even workload across all the outgoing routes to the servers.

6.2.2.3 EVENT MESSAGES

Each event message used in the model keeps track of the client from which it was originally sent. This allows the server to create a response message which will return to the original client. The response messages are forwarded back along a route to the correct client group incoming gateway, and then the incoming gateway will forward the message to the correct client.

6.2.2.4 NETWORK ROUTES

There are two sets of network routes which form pipe tasks between the client groups and the servers. Each pipe task has one entry point, and one exit point. Each pipe task is uni-directional, so messages are sent to the server through a different pipe task than they return on. An outgoing message entering the task will be forwarded through all LPs in the task, before reaching the final LP that will randomly pick a server to send the message on to. The network routes allow time parallelism which should increase the performance of the model.

6.2.2.5 SERVERS

Each server exists in a single task by itself. It is expected that the servers will have the heaviest load during the simulation, depending on their number, and having each in a task will allow for maximum parallelisation. When a server receives a request it performs some arbitrary processing, attempting to model the work a real server might do, before sending a response back out across a network route.
6.2.3 Arbitrary Work

In both models it is mentioned that some arbitrary work can be performed, this takes the form of calculating Pi to a given number of digits based on the code made available by Angsuman Chakraborty [100]. The sample code was modified slightly to make the static methods and variables instance methods and variables. This was necessary to prevent many LPs using the same instance of the class at the same time, akin to using shared variables.
7 TESTING

Testing occurred throughout the development process, and all errors that were encountered were debugged. Developing the simulation models only goes so far in proving the correctness of the implementation. Comprehensive logging can be enabled to track system actions which can provide further evidence for correctness. The logs analysed in this section are available in Appendix A, with key lines marked in **bold**.

In order to capture the full range of possible events in the simulation, a number of modified models based on those described in section 6.2 are used. The test scenarios, as well as these modifications are discussed in the following sections.

All tests are performed on a Windows XP Professional desktop PC with an Intel Q6600 Quad Core CPU @ 3.2Ghz, 2Gb DDR2 Memory @ 800Mhz, and running Java 2 SE 6.0. Java is an established language, and the APIs are thoroughly tested against all applicable platforms so we can be reasonably confident results are not machine/platform dependent.

7.1 SOURCE LP EXECUTION

When a source LP is executed it should generate events with timestamps up to the current LP clock value + sourceLPRunningTime and then reschedule itself and the task it is a member of. The output channel clock should be updated to the clock of the source LP + the channel delay value. The Pipe Model is executed, with a sourceLPRunningTime value of 2 and an output channel delay of 10.

Appendix A.1 shows the relevant section of the output log, and demonstrates correct behaviour.

7.2 NON-SOURCE LP EXECUTION

When a non-source LP is executed, the input channels should be polled for events, and where found added to the event queue. Events from the event queue are then executed up to the safe time, and source channels are polled for more events as necessary. When an execution session finishes, the clocks should be updated, and the input channel which defines the safe time should be marked critical.

A slightly modified version of the Pipe Model is used, which includes two Sender LPs, which both send events to a single receiver LP. The sourceLPRunningTime is set to 2, and the channel delays are set to 10.

Appendix A.2 shows the relevant section of the output log, and demonstrates correct behaviour. Note that the logging for the sender LPs is omitted for brevity.
7.3 Scheduling Internal Critical Channel

If an LP executes, and as a consequence of that execution an output channel that is flagged as critical, advances its clock value, the destination LP for that channel should be scheduled for execution. If the channel is internal, then the destination LP should be rescheduled within the task. The Pipe Model is used, with a sourceLPRunningTime of 2, and channel delays set to 10. The important thing to note in this test is that the scheduling is performed locally, in the same task.

Appendix A.3 shows the relevant output log section for this test, and demonstrates correct behaviour.

7.4 Scheduling External Critical Channel

This test is the same as the above, except that the two LPs are in different tasks. This means that when the destination LP is scheduled on a critical channel the LP and its parent task should be scheduled. A slightly modified version of the Pipe Model is used, which has each LP in a separate task.

Appendix A.4 shows the relevant output log section, and demonstrates correct behaviour.

7.5 Executing Events Up To The Safe Time

When an LP with multiple input channels is executing, events should not be processed which have timestamp values which are greater than the safe time for that LP. The safe time is refined down as empty channels are encountered. A modified Pipe Model is used which has two sender LPs, and one receiver LP. One sender LP works as normal, and one sends an extra event with a timestamp 10 higher. The receiver LP should process events from both input channels until an empty channel is encountered. The high timestamp event should not get processed as the safe time value is refined down by the empty channel.

Appendix A.5 shows the relevant output log section, and demonstrates correct behaviour. The important thing to note in the log is that three events are received from the input channels, but only two are processed.

7.6 Receiving Events From Channels

When an LP is executing, it should only take events from channels which it currently does not already have an event from in the event queue. To test this, a modified version of the Simple Pipe model is used, which has two senders. One sender behaves as normal, and one sender
sends events with higher timestamps. The LP should take the first high timestamped event on the initial execution phase, and since this event will not be processed it should not take another one the next phase.

Appendix A.6 shows the relevant log section, and demonstrates correct behaviour. The important thing to note is that the first time the receiver LP executes, it removes an event from both input channels, and from only one on the second execution.

### 7.7 Discarding Events Past The Simulation End Time

Event messages generated which have timestamp values greater than the simulation end time. For this test the Simple Pipe model is used the simulation end time is set to 1, and the channel delays are set to 10. The first event generated by the sender will have a timestamp of 1, the channel delay will mean the channel will not accept the event.

Appendix A.7 shows the relevant log section, and demonstrates correct behaviour. Note that since the channel is responsible for rejecting the event that the sending LP still sends the event message.

### 7.8 Self Scheduled Events

An LP can schedule events for itself which are added to the LP’s event queue. A modified version of the Pipe Model is used in which the receiver reschedules received events for itself, and increases the timestamp of that event by 1. The log should show an LP receiving events, scheduling self events, and processing these self events until the safe time is reached.

Appendix A.8 shows the relevant output log section, and demonstrates correct behaviour.
8 RESULTS

This section contains the performance results of testing the simulation kernel using the two models described in section 6.2. The performance of the system is tested as the number of available processors is increased, as well as event granularity, and simulation scale. Across all tests what we are interested in is the speedup seen as more processors are available, and are not aimed to provide an exact benchmark of the simulation kernel, but as an investigation into the impact of influencing factors of simulation speed.

There are very many parameters in the simulation models and kernel which can be varied to collect performance results. For each set of tests performed all parameters except those explicitly specified remain the same to isolate the effects of the parameters which are changed.

All tests are performed on a Windows XP Professional desktop PC with an Intel Q6600 Quad Core CPU @ 3.2Ghz, 2Gb DDR2 Memory @ 800Mhz, and running Java 2 SE 6.0. Java is an established language, and the APIs are thoroughly tested against all applicable platforms so we can be reasonably confident results are not machine/platform dependent. JVM parameters are used to increase the maximum memory size to 1 GB.

8.1 OBSERVED SPEEDUP

I intend to measure performance using a variant of observed speedup, a simple and popular indicator for parallel program performance, defined as:

\[
\text{wall clock time of serial execution / wall clock time of parallel execution.}
\]

Since this formula only measures speedup compared to sequential computation I will extend it to indicate performance change with variable processor counts.

\[
\text{wall clock time of x processor execution / wall clock time of y processor execution.}
\]

Where \( X < Y \), to give the relative speedup obtained using a greater number of processors.

Timing results are obtained programmatically using built in Java timing API and are the average of 5 test runs. Note that results are accurate to the nearest 50ms due to the termination detection mechanism used. Full result tables are listed in Appendix 2.

For each simulation run the simulation end time is adjusted so that the total wall clock time elapsed for each simulation is as close as possible; it is not practically possible to exactly balance the running times so minor variances are seen. This should not affect the results in any significant
way, and since we are interested mostly in relative performance, rather than absolute values, are allowed to remain.

**8.2 Simulation Scale**

The scale of the simulation affects the amount of work required overall to progress the simulated time of the system. In this section performance results are collected which show the speedup gained using multiple processors for variously sized Request Response models. For all tests the number of routes between clients groups and servers and the return route is kept the same.

**8.2.1 Small Request Response Model**

The results below show a very slight slowdown in performance as more processors are added and are probably due to the overheads incurred with thread management. Adding more processors also decreases the number of events which are processed during each execution phase at an LP which may be another contributing factor. Hopefully the results will improve as the simulation size is scaled upwards.

One possible cause for the poor performance in such a small simulation is that contention for access to LPs is causing destructive interaction between threads. Simulations of this scale may effectively fit into processor caches, and the performance encountered could be a result of unnecessarily sharing work across multiple processors, resulting in inefficient execution and poor cache locality.

<table>
<thead>
<tr>
<th>Group Count</th>
<th>Group Size</th>
<th>Route Count</th>
<th>Route Length</th>
<th>Servers</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
8.2.2 MEDIUM REQUEST RESPONSE MODEL

The results below show an almost linear speedup when using 2 and 3 processors, but after that the performance drops off slightly. It appears that the simulation has reached a size in which parallelism becomes an effective performance boost for smaller processor counts, but the algorithm is still a limiting factor for four processors. The speedup seen is still small, with a maximum speed up of approximately 1.4 using 3 processors.

The reduction in speed up seen for four processors is caused by a limit being reached to the benefit of adding more processors. As the simulation scale increases more work is available in total, and across a greater number of LPs and tasks. The more ‘space’ each thread has to execute, without meeting other threads vying for the same work, the more efficiently the hardware resources can be utilised.

<table>
<thead>
<tr>
<th>Group Count</th>
<th>Group Size</th>
<th>Route Count</th>
<th>Route Length</th>
<th>Servers</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
</tr>
</tbody>
</table>
8.2.3 LARGE REQUEST RESPONSE MODEL

The results below show an almost linear speedup up to four processors. It appears that the simulation has reached a size in which parallelism becomes an effective performance boost. Further testing with higher processor counts could be used to see if the trend continues. A maximum speed up of approximately 1.4 is reached with 4 processors, this is the same as was achieved using 3 processors in the medium scale model.

This is the first simulation in which adding additional processors has not decreased the speed up achieved. The simulation scale has reached a level where four threads can be used to execute the model without causing more destructive interaction than the benefit gained by their inclusion.

<table>
<thead>
<tr>
<th>Group Count</th>
<th>Group Size</th>
<th>Route Count</th>
<th>Route Length</th>
<th>Servers</th>
</tr>
</thead>
<tbody>
<tr>
<td>25</td>
<td>25</td>
<td>25</td>
<td>25</td>
<td>25</td>
</tr>
</tbody>
</table>
8.3 Event Granularity

Event granularity is the high-level measure of work associated with each event message. Low granularity events require less processing than high granularity events. These tests are aimed to measuring the speedup achieved by using different granularity events.

The Request Response Model is used to perform these tests, and is modified slightly so that each LP includes some arbitrary processing at each LP per event processed. The scale of the simulation is set to the same as the medium scale simulation from the simulation scale tests.

8.3.1 Very Low Granularity Events

The speedup seen is minimal (approximately 1.3), and adding additional processors has not had any affect. These results are similar to the medium scale simulation results above. The increase from one processor to two processors has increased performance by the largest amount.

<table>
<thead>
<tr>
<th>Group Count</th>
<th>Group Size</th>
<th>Route Count</th>
<th>Route Length</th>
<th>Servers</th>
<th>Digits of Pi to Calculate / Event</th>
</tr>
</thead>
<tbody>
<tr>
<td>25</td>
<td>25</td>
<td>25</td>
<td>25</td>
<td>25</td>
<td>0</td>
</tr>
</tbody>
</table>
8.3.2 Low Granularity Events

Low granularity events show a much increased speedup per processor than very low granularity events with a maximum speedup of approximately 1.8. Speedup continues to improve after each additional processor is added. However, the amount of speedup seen per processor decreases with each one added. Diminishing returns are gained, with the fourth processor creating much less of a performance increase than the previous two.

The diminishing returns encountered could be caused by destructive interactions between threads, as discussed in section 8.2. The absolute speedup seen is higher than for very low granularity events, and it is likely that as event granularity increases a greater proportion of processing time will be spent executing arbitrary work, and leave fewer opportunities for destructive interaction.

<table>
<thead>
<tr>
<th>Group Count</th>
<th>Group Size</th>
<th>Route Count</th>
<th>Route Length</th>
<th>Servers</th>
<th>Digits of Pi to Calculate / Event</th>
</tr>
</thead>
<tbody>
<tr>
<td>25</td>
<td>25</td>
<td>25</td>
<td>25</td>
<td>25</td>
<td>10</td>
</tr>
</tbody>
</table>
8.3.3 High Granularity Events

High granularity events show a significant increase in speedup with each processor added. In particular the fourth processor added results in a greater speedup than seen previously.

<table>
<thead>
<tr>
<th>Group Count</th>
<th>Group Size</th>
<th>Route Count</th>
<th>Route Length</th>
<th>Servers</th>
<th>Digits of Pi to Calculate / Event</th>
</tr>
</thead>
<tbody>
<tr>
<td>25</td>
<td>25</td>
<td>25</td>
<td>25</td>
<td>25</td>
<td>100</td>
</tr>
</tbody>
</table>
8.3.4 Very High Granularity Events

An almost linear speedup is shown up to three processors. The maximum speedup shows of nearly 3.0 is a large increase on previous results and evidences that larger granularity events improve the speedup achieved in the simulation.

It can be taken from these results that as event granularity increases the computation overhead of the algorithm becomes much smaller in relation to the work performed by the model. It could be expected that in further tests, with ever greater amounts of arbitrary work performed at each event, that the speedup achieved would gradually converge with the maximum possible according to Moore’s law.

<table>
<thead>
<tr>
<th>Group Count</th>
<th>Group Size</th>
<th>Route Count</th>
<th>Route Length</th>
<th>Servers</th>
<th>Digits of Pi to Calculate / Event</th>
</tr>
</thead>
<tbody>
<tr>
<td>25</td>
<td>25</td>
<td>25</td>
<td>25</td>
<td>25</td>
<td>1000</td>
</tr>
</tbody>
</table>
8.3.5 Event Granularity Overall Graph

Figure 8.7 Speedup graph for very high granularity event test.

Figure 8.8 Event granularity speedup comparison graph.
## 8.4 Time Parallelism

The simple pipe model is used to demonstrate speedup due to time parallelism. A single pipe task is used, with an LP count equal to the maximum number of processors being used to test with. At each LP some arbitrary work is performed.

The results show a significant speedup, evidencing that the time parallelism in the system is being utilised by the simulation. There is a declining increase in performance gained as the processor count is increased, caused by the limits to time parallelism in the system (see section 9.4). In a typical model where, the number of tasks out numbers the number of available processors, it would not be expected for this limit to significantly affect performance.

![Speedup Graph](image)

**Figure 8.9 Time parallelism speedup results graph.**

### 8.5 Thread Blocking

Thread blocking occurs when synchronised access to some resource is required by multiple threads at the same time. In the simulation kernel there are a few opportunities for this to arise:

![Thread Blocking Graph](image)

**Figure 8.10 JProfiler thread monitor graph. [Yellow] = Sleep, [Red] = Blocked.**
1. The central task queue is thread safe, which requires some blocking in order to synchronise item queuing and de-queuing operations.

2. When an LP finishes an execution phase and attempts to schedule the destination LP for a critically flagged outgoing channel the thread blocks until that channel is not flagged as busy.

Thread blocking can be analysed by a profiling tool, Jprofiler [101] profiling tool was used to analyse the execution of the Request Response model. Figure 8.10 JProfiler thread monitor graph. [Yellow] = Sleep, [Red] = Blocked. shows the Thread monitor graph for just under a minute of execution.

The graph shows an initial initialisation period up until about four seconds where the main thread is occupied, and the thread pool has not started up yet. At around 4 seconds the thread pool is initialised, and for around a half second after that several threads are blocked briefly. Throughout the simulation blocking is encountered in clusters, probably occurring when one part of the simulation becomes a bottleneck to other parts and threads are forced to block for it to catch up. Figure 8.11 shows the processor utilisation graph for the same time period and shows a correlating drop in utilisation during the blocking sections.

Once the initialisation period is over the main thread is responsible only for detecting the simulation end time but shows a lot of activity. This is likely due to the inefficient termination detection algorithm in place and would need to be improved to avoid becoming a bottleneck in larger scale simulations.
Figure 8.11 Processor utilisation graph for thread blocking analysis.

### 8.6 MEMORY USAGE

Jprofiler was used to analyse memory usage throughout the simulation, Figure 8.12 shows the memory usage history graph for a large scale Request Response simulation and shows a steep upwards trend until the maximum 1024 MB of memory allocated to the JVM is exhausted.

Figure 8.13 captures the same information during execution of a much smaller simulation model. As expected memory is consumed less quickly, and interestingly the memory footprint of the simulation reaches a relatively steady level at around 115 megabytes ~7 seconds into the simulation. It is possible that on a machine with more memory the steady state found in this simulation would be reached in larger simulations as well. The causes of the large memory spikes may be attributed to a Server LP executing, which requires more memory. The memory used is quickly garbage collected.
Figure 8.12 JVM memory usage during a large scale simulation run.
8.7 PROCESSOR TIME

JProfiler can be used to analyse the time spent in each section of code by processors during simulation execution. The request response model was executed, with no arbitrary work performed so that the algorithm mechanisms would be stressed as much as possible. Figure 8.14 shows that the largest proportion of execution time is spent fetching and comparing event message timestamps. This is to be expected since accessing timestamps is by far the most frequent operation in the simulation. Nearly all of the execution hotspots identified by the profiler are atomic operations which either store or retrieve a value. Improving the performance of the simulation will be difficult, as the execution speed of these operations is largely down to the Java Virtual Machine.
A Simulation Kernel

<table>
<thead>
<tr>
<th>Hot spot</th>
<th>Inherent time</th>
<th>Average Time</th>
<th>Invocations</th>
</tr>
</thead>
<tbody>
<tr>
<td>Event.getEvent()</td>
<td>27,201 ms (10%)</td>
<td>3 μs</td>
<td>8,527,591</td>
</tr>
<tr>
<td>EventComparator.compare(Comparator, Comparator)</td>
<td>24,386 ms (9%)</td>
<td>9 μs</td>
<td>2,592,186</td>
</tr>
<tr>
<td>Task.getTaskClock()</td>
<td>22,100 ms (8%)</td>
<td>215 μs</td>
<td>102,424</td>
</tr>
<tr>
<td>LogicalProcess.executeEvents</td>
<td>19,491 ms (7%)</td>
<td>7.147 μs</td>
<td>2,727</td>
</tr>
<tr>
<td>Channel.sendEvent()</td>
<td>18,090 ms (6%)</td>
<td>20 μs</td>
<td>618,251</td>
</tr>
<tr>
<td>EventComparator.compare(java.lang.Object, java.lang.Object)</td>
<td>15,044 ms (5%)</td>
<td>5 μs</td>
<td>2,592,186</td>
</tr>
<tr>
<td>LogicalProcess.scheduleExternalEvent</td>
<td>12,576 ms (4%)</td>
<td>20 μs</td>
<td>618,251</td>
</tr>
<tr>
<td>java.util.PriorityQueue.remove</td>
<td>9,394 ms (3%)</td>
<td>15 μs</td>
<td>618,251</td>
</tr>
<tr>
<td>java.util.concurrent.ConcurrentLinkedQueue.add</td>
<td>9,168 ms (3%)</td>
<td>14 μs</td>
<td>618,251</td>
</tr>
<tr>
<td>Channel.getEvent()</td>
<td>8,676 ms (3%)</td>
<td>17 μs</td>
<td>497,973</td>
</tr>
<tr>
<td>LogicalProcess.doClock()</td>
<td>7,834 ms (2%)</td>
<td>2 μs</td>
<td>2,711,222</td>
</tr>
<tr>
<td>java.util.Iterator.hasNext</td>
<td>7,599 ms (2%)</td>
<td>2 μs</td>
<td>2,630,902</td>
</tr>
<tr>
<td>java.util.Iterator.next</td>
<td>7,375 ms (2%)</td>
<td>2 μs</td>
<td>2,519,574</td>
</tr>
<tr>
<td>LogicalProcess.getEventQueue</td>
<td>6,209 ms (2%)</td>
<td>2 μs</td>
<td>1,988,670</td>
</tr>
<tr>
<td>NetworkNode.processEvent</td>
<td>3,713 ms (1%)</td>
<td>10 μs</td>
<td>368,621</td>
</tr>
<tr>
<td>Channel.getDelay()</td>
<td>3,675 ms (1%)</td>
<td>3 μs</td>
<td>1,116,697</td>
</tr>
<tr>
<td>java.lang.StringBuilder.append(java.lang.String)</td>
<td>2,661 ms (1%)</td>
<td>21 μs</td>
<td>128,023</td>
</tr>
<tr>
<td>java.util.PriorityQueue.add</td>
<td>2,611 ms (1%)</td>
<td>4 μs</td>
<td>618,705</td>
</tr>
<tr>
<td>Client.processEvent()</td>
<td>2,410 ms (1%)</td>
<td>19 μs</td>
<td>125,000</td>
</tr>
<tr>
<td>LogicalProcess.scheduleSelfEvent</td>
<td>2,362 ms (1%)</td>
<td>18 μs</td>
<td>125,000</td>
</tr>
<tr>
<td>Event.setTime()</td>
<td>2,285 ms (1%)</td>
<td>3 μs</td>
<td>743,705</td>
</tr>
<tr>
<td>LogicalProcess.doSimulationEndTime</td>
<td>2,216 ms (1%)</td>
<td>3 μs</td>
<td>749,294</td>
</tr>
<tr>
<td>Event.setChannel()</td>
<td>2,173 ms (1%)</td>
<td>3 μs</td>
<td>618,619</td>
</tr>
<tr>
<td>Channel.waitForLastEventTime</td>
<td>2,132 ms (1%)</td>
<td>5 μs</td>
<td>618,619</td>
</tr>
<tr>
<td>java.util.Arrays.asList()</td>
<td>2,124 ms (1%)</td>
<td>19 μs</td>
<td>111,227</td>
</tr>
<tr>
<td>Event.getChannel()</td>
<td>2,069 ms (1%)</td>
<td>3 μs</td>
<td>618,621</td>
</tr>
<tr>
<td>Channel.getEventQueue()</td>
<td>2,072 ms (1%)</td>
<td>3 μs</td>
<td>622,887</td>
</tr>
<tr>
<td>Channel.getChannel()</td>
<td>2,051 ms (1%)</td>
<td>3 μs</td>
<td>623,036</td>
</tr>
<tr>
<td>Process.getProcess()</td>
<td>2,049 ms (1%)</td>
<td>3 μs</td>
<td>618,621</td>
</tr>
<tr>
<td>java.util.HashMap.get()</td>
<td>2,047 ms (1%)</td>
<td>3 μs</td>
<td>629,802</td>
</tr>
<tr>
<td>java.util.concurrent.ConcurrentLinkedQueue.peek</td>
<td>2,040 ms (1%)</td>
<td>3 μs</td>
<td>621,244</td>
</tr>
<tr>
<td>Channel.getEventQueue()</td>
<td>2,029 ms (1%)</td>
<td>3 μs</td>
<td>618,991</td>
</tr>
<tr>
<td>java.lang.Message&lt;br&gt;Queue.add</td>
<td>1,937 ms (1%)</td>
<td>7 μs</td>
<td>256,182</td>
</tr>
<tr>
<td>Event.&lt;init&gt;()</td>
<td>1,903 ms (1%)</td>
<td>7 μs</td>
<td>250,000</td>
</tr>
<tr>
<td>java.util.concurrent.ConcurrentLinkedQueue.poll</td>
<td>1,802 ms (1%)</td>
<td>3 μs</td>
<td>467,973</td>
</tr>
<tr>
<td>java.lang.Process.println</td>
<td>1,600 ms (1%)</td>
<td>2.725 μs</td>
<td>587</td>
</tr>
<tr>
<td>OutgoingGroupRouter.processEvent</td>
<td>1,559 ms (1%)</td>
<td>12 μs</td>
<td>125,000</td>
</tr>
</tbody>
</table>

Figure 8.14 Processor time hot spots in the simulation kernel.
9 Evaluation & Conclusion

Throughout the duration of this project it has been possible to learn a great deal about computer simulation (especially parallel simulation), thread management, and access synchronization. The project has also been an excellent exercise in project planning, scientific research, software design, implementation, testing, and evaluation. The entire experience has been both extremely challenging and greatly rewarding.

This section evaluates the project outcome against the original project aims, critiquing the results obtained, and the processes used throughout. Future work is outlined as possible extensions to this project, and as a way to publish the simulation kernel developed alongside those already publically available.

9.1 Project Aims

The project aims were met, an implementation of the CCT algorithm was produced as part of a parallel simulation kernel. Results relating to the performance of the simulation were collected by developing and running two simulation models with various parameters. Correctness of the implementation was evidenced by executing the two simulation models and analysing log output under various circumstances.

9.2 Results

The performance results collected are positive, showing a maximum speed up of approximately 3.0 using four processors on a medium scale, high granularity simulation model. However, using small granularity events the performance gains are much smaller. Further testing is required to assess performance with larger numbers of processors and different simulation models. The performance results obtained demonstrate expected behaviour based on the changing of parameters discussed in section 4.4.

The University of Bath makes available a 16 processor Sun Microsystems server running Java 1.5.0 on which it was hoped larger processor counts could be used for further testing. However, during testing the Simulation class was throwing ConcurrentAccessException errors at seemingly random times. An attempt was made to debug the error, but it could not be tracked down to any of the Simulation code, and seemed to be caused by the PriorityBlockingQueue being used. Usable results could not be collected, and unfortunately another machine with more than four processors could not be found.
9.2.1 Simulation Scale

The simulation scale results show an improving speed up as the simulation size is increased. This is because as more work is available for a set number of processors it is more likely that the processors can operate independently and without negative interactions. The increase in performance seen when scaling up the simulation begins to decrease after a certain level; this may be due to the increased memory requirements of the simulation. Memory is likely to be the first bottleneck in a CCT based simulation on a shared memory machine.

9.2.2 Event Granularity

Event granularity has the biggest impact on simulation performance out of the factors tested. This is because event granularity directly shifts processing time away from algorithm mechanics and into model based work; such a shift allows simulation time to advance more efficiently.

9.3 Improving Performance

9.3.1 Memory Usage

Memory usage in the implementation of the CCT algorithm was identified as a possible problem in section 2.4.2.7.1, and during testing it proved to be a limiting factor in the scale of simulation possible. The high memory usage was caused by a massive number of event messages being generated by the system and pooling in the channel queues. The cause of the high memory usage is likely one of the cases identified earlier, where a slower LP (such as a server in the case of the Request Response model) is unable to keep up with the number of events arriving at its input channels, leading to the input channel event queues growing unchecked.

Limiting buffer usage should smooth out model execution speed, and prevent the “blocking sections” seen in the thread activity results (see Figure 8.10), by preventing very large queues of events to build up at any LP.

9.3.2 Memory Management

Performance could very likely be improved with an increased understanding of the low level workings of Java and the JVM. In a system which is highly dependent on efficient memory allocation and cache usage across multiple processors, a lower level of control over these aspects would enable a more efficient implementation. This would undoubtedly come at a cost of increased complexity, reduced readability, and may decrease the portability of the system if architecture/OS specific optimisations are implemented.
9.3.3 REMOVING GETTER AND SETTER METHODS

The getter and setter methods currently implemented for accessing commonly used fields are causing an extra method call for each such operation. It may break OO design principles to do so, but allowing direct access to fields will improve performance. This improvement, although small, could benefit performance greatly as in one minute of normal execution around twelve million calls were made to methods requesting class variables.

9.3.4 LANGUAGE CHOICE

Java was justified as the implementation language in section 6.1.1, however, and in making this decision a compromise was made on the performance of the system. Languages such as C++ have consistently demonstrated better performance than Java [102] on a multitude of different operations.

One way to specifically improve performance of the system would be to re-implement it in a faster language such as C or C++. However, it may be possible to borrow some of the desirable features of C++ in order to increase running speed. There are some Java Compilers available, such as GCJ [103], which allow Java byte code to be converted to native machine code in Linux, often realising significant performance improvements. The Java JIT (Just in Time) compilation introduced in Java 6 includes some support for compiling into native instructions, but is still being optimised, and as yet has failed to reach speeds achieved by natively compiled C++ code.

9.3.5 SIMULATION INITIALISATION

The initialisation stage of the simulation is currently not parallelised. This stage of execution, where the model is built, normally takes a few seconds depending on the simulation size. It is currently possible for the simulation modeller to implement their own multi-threaded initialisation procedure; however, providing such a mechanism in the system would be beneficial to the user. This could be achieved by providing some kind of thread factory which the simulation modeller can call on to spawn new threads.

9.3.6 MODEL PERFORMANCE ANALYSIS

Simulation performance may be improved by providing the model developer with more useful tools for execution analysis. Providing information about the performance of individual LPs, such as LPs which use significantly more CPU time than other, or identifying high traffic areas of simulations may allow the modeller to adjust the models, and increase performance.

This could be implemented if each base LP were to record performance data throughout the execution of a simulation such as; events processed per second and simulated time progressed per second. Such statistics could be reviewed by the modeller and used to identify bottlenecks in the model.
9.3.7 NON-UNIFORM MEMORY ACCESS (NUMA)

NUMA architectures explicitly separate and allocate sections of system physical memory to processors. This architecture has showed significant memory bandwidth improvements, and with the frequent memory manipulations encountered in a simulation could prove a significant performance boost. The downside of such architectures is that they require specialised hardware, which can be expensive.

NUMA architectures draw similarities with cluster or grid computers, as each processor has a dedicated memory interface. However, unlike cluster or grid computers the physical separation of components is minimal (they are all part of the same machine), meaning that communication between processors is relatively fast compared with even the fastest network.

9.3.8 JVM FLAG OPTIMISATION

I mentioned in section 7 that a JVM flag was used to specify the maximum amount of system memory available to the JVM during execution. Further flags are available for pre-allocating the maximum available memory to the JVM before execution, and can improve performance by eliminating costly memory request operations during run-time, and also by allocating a more contiguous memory block to the JVM.

In applications which require a lot of memory, the Java garbage collector (GC) can require significant processing resources. When the GC runs, it pauses all running threads and executes using a single thread. This clearly wastes the processing resources available on a multi-processor machine, and can be avoided by using the Java parallel garbage collector via the `-XX:UseParallelGC` JVM flag.

The `-Xmn<size>` JVM flag allows manual specification of the "young generation" object memory space size. During a simulation many objects are created which have short lifetimes, and it may be possible to improve performance by experimenting with larger values of this flag.

9.4 REQUIREMENTS AND DESIGN

The requirements for the system are largely based around the description of the CCT algorithm in [1], and the identification of additional requirements was a difficult process. The simulation kernels that are available are either based on sequential algorithms, or implemented in a lower level language such as C++. In designing the requirements for this project, the usability of the system was important, and the design criteria reflect this. The system is simple enough for someone with some understanding of the CCT algorithm and some programming experience to understand. This was one of the aims of the 'Simple' design criteria.
The design was created incrementally by adding additional functionality onto the framework required by the algorithm. This seemed to work very well, although for a more complex system, an initial design which considered all intended functionality from the start would have been needed. Some problems with the design did not become apparent until the implementation stage, such as the need for duplicating data within each LP to avoid using shared variables. Such problems were solved as they were encountered, and documented in the design and implementation section.

Model design and implementation was another important aspect of this project, and ties into the usability of the system. The system does not require the model builder to understand the specifics of the simulation kernel, but does require some familiarity with the complex topic of distributed simulation model design.

One problem with the current design is how time parallelism is handled. It is possible for the first LP in a pipe task to finish execution, and the current thread to move to the second LP in the task. Whilst the first thread is processing events waiting at the second LP, it is possible for another thread to enter the pipe and generate more events for this second LP. This will cause the first thread to continue processing events at the second LP until the events from both executions of the first LP are executed. When the second thread finishes execution of the first LP, it will have to wait for the first thread to finish the second LP. At this time there will be no events waiting at the second LP. Such a process can continue the full length of the pipe task, with one thread essentially doing twice as much work, and one spending a lot of time blocking. It is possible that the blocking shown in the thread activity graph in the results section is caused by such a situation.

One possible solution to the above problem would be to save the clock time of input channels to an LP when it begins executing, and only process events up to this clock time, ignoring events which enter the channel after this. Such a solution is not ideal and goes against one of the main aims of the CCT algorithm - to process as many events at each LP as possible.

9.5 Implementation

The implementation followed the same stages as the design, being built in stages based on the next phase of functionality required. Some refactoring was required at each stage, which in turn required more testing, but on the whole the implementation proceeded smoothly with one notable exception (see section 9.5.2).

The Eclipse IDE provides functionality to help with refactoring; these automated tools can be used for renaming variables, splitting classes into sub classes, and generating getter and setter methods. Using these tools reduced the risk of introducing bugs as development progressed.
9.5.1 Running a New Simulation Model

The method for running a new model using the simulation kernel requires that the user edit and recompile the `Main.java` file. This is an inconvenience to the user, and ideally would not be required. It may also limit the possible user base of the kernel to those with the Java 1.6.0+ Development Kit installed, as this is required to compile the `Simulation` class.

A better approach would be to process the Simulation Model Name parameter in some more useful way, and automatically attempt to load a class of this name during runtime. The user would still be required to specify the class path to their simulation model, but this can be considered a minor inconvenience as opposed to the current situation.

9.5.2 The Missing Event Bug

The missing event bug caused some events in the system to ‘disappear’, they were not executed, and were not passed on in the system. The bug was encountered during testing of the simple pipe model when asserting that the number of events generated by the sender LP was the same number of events received by the receiver LP. It was not immediately apparent where the bug was, and there are many parts of the system which handle events and needed to be checked.

To track down the cause of the bug, the Eclipse Java debugger was used to step through the code, and analyse the behaviour of an LP during the event execution phase. The bug was eventually found to be caused by an incorrect loop during event execution, which fetched the next event from the LP event queue, and then compared the timestamp with the current safe time. If the event timestamp was greater than the safe time then the event was not executed, however, it was also not being put back into the event queue. This problem was fixed by using the `peek()` method of the event list to look at the next event without removing it from the queue.

9.5.3 Termination Detection

The termination mechanism developed is very processor intensive, a problem which only worsens as the simulation scale is increased. The current mechanism is wasteful in that it loops through all tasks and LPs in the simulation to determine the least progressed LP.

A more robust and efficient mechanism would be to extend the `ThreadPoolExecutor` class to detect when the task queue becomes empty, and there are no LPs currently being executed. As long as one LP is executing more work may be scheduled by it, meaning that an empty task queue alone cannot indicate that the simulation has terminated. This mechanism could also be used to detect simulations which end before the simulation end time is reached, something which currently is not possible.


9.6 Possible Extensions

The project could be extended in a number of ways. Firstly more results need to be obtained which test the simulation when executed on more processors, different simulation models, and possibly on different processor architectures.

The implementation can be extended to include different task types other than the pipe task, such as cluster tasks as identified in [1]. The foundations for new task types are already in place, tasks could be implemented by extending the current Task class and overriding the methods which determine the execution and scheduling of LPs.

A GUI that allowed visualisation of LP progress, CPU utilisation and message activity would be a great tool for the simulation model developer to analyse model performance and identify bottlenecks. Such a GUI would require hook in points in the main classes to send updates to the GUI periodically in order to update statistics such as queue size, events processed/sec, etc.

To combat the high memory usage seen with simulations, one of the buffer management schemes developed in [28] could be added to the current implementation to improve performance and scalability. Such additions would require substantial work, as they require alterations to the core algorithm and may produce knock-on effects which would require re-testing.

9.6.1 Comparison with Java SimKit

The SimKit [68] Java simulation API currently only supports a sequential execution model. The kernel developed for this project could be modified to provide the same model interface as the existing SimKit API and allow existing models developed for SimKit to be executed in parallel. Such an extension would require significant work, but with the popularity of SimKit would provide greater exposure for the system.

9.7 Summary

This project has demonstrated that effective speedup can be achieved using multiple processors to execute a CCT based simulation kernel. The investigation into the effects of simulation scale and event granularity on performance returned results which correlate both with the theory of expected speedup, as well as other studies into their effect on similar systems.

It is hoped that the software produced is powerful enough to execute any arbitrarily complex simulation model, whilst remaining easy to use, and providing a solid core of features which can be easily extended if required.
10 BIBLIOGRAPHY


A Simulation Kernel


A Simulation Kernel


A Simulation Kernel


64. Richard, E.N., A history of discrete event simulation programming languages, in The second ACM SIGPLAN conference on History of programming languages. 1993, ACM: Cambridge, Massachusetts, United States.


74. Merrifield, B.C., S.B. Richardson, and J.B.G. Roberts. Quantitative studies of discrete event simulation modelling of road traffic. in SCS Multiconference on Distributed Simulation. 1990.


92. Microsystems, S. *PriorityBlockingQueue (Java 2 Platform SE 5.0)*. [cited 25/03/08]; Available from: http://java.sun.com/j2se/1.5.0/docs/api/java/util/concurrent/PriorityBlockingQueue.html.

93. Microsystems, S. *Queue (Java 2 Platform SE 5.0)*. [cited 25/03/08]; Available from: http://java.sun.com/j2se/1.5.0/docs/api/java/util/Queue.html.

94. Microsystems, S. *Comparable (Java 2 Platform SE 5.0)*. [cited 25/03/08]; Available from: http://java.sun.com/j2se/1.4.2/docs/api/java/lang/Comparable.html.

95. Microsystems, S. *LinkedList (Java 2 Platform SE 5.0)*. [cited 25/03/08]; Available from: http://java.sun.com/j2se/1.5.0/docs/api/java/util/LinkedList.html.

96. Microsystems, S. *Thread (Java 2 Platform SE 5.0)*. [cited 25/03/08]; Available from: http://java.sun.com/j2se/1.4.2/docs/api/java/lang/Thread.html.


98. Microsystems, S. *PriorityQueue (Java 2 Platform SE 5.0)*. [cited 25/03/08]; Available from: http://java.sun.com/j2se/1.5.0/docs/api/java/util/PriorityQueue.html.


Appendix A: Testing Log Outputs

1. Source LP Execution

Simulation Started  
Task@1d5550d Task Started Execution  
Task@1d5550d Task Copying Startup Scheduled LPS, Found: 4  
Task@1d5550d Starting Execution of LP: SimpleSender@c21495  
SimpleSender@c21495 [8] :: Starting Execution  
SimpleSender@c21495 [8] :: Polling Input Channels for Events  
SimpleSender@c21495 [8] :: Source LP Generating Events  
[SimpleSender@c21495 -> SimpleForwarder@c2ea3f] :: Adding Event: Frame@1833955, with Time: 0  
[SimpleSender@c21495 -> SimpleForwarder@c2ea3f] :: Adding Event: Frame@186db54, with Time: 1  
[SimpleSender@c21495 -> SimpleForwarder@c2ea3f] :: Adding Event: Frame@21b6d, with Time: 2  
SimpleSender@c21495 [8] :: Frames Sent: 3  
SimpleSender@c21495 [8] :: Updating Clock To: 2  
[SimpleSender@c21495 -> SimpleForwarder@c2ea3f] :: Updating Clock To: 12  
SimpleSender@c21495 [8] :: Source LP Rescheduling Self Globally  
SimpleSender@c21495 [8] :: Now Scheduled Globally  
Simulation Scheduling Task: Task@1d5550d  
SimpleSender@c21495 [8] :: Finished Execution

2. Non-Source LP Execution

Task@c21495 Starting Execution of LP: SimpleReceiver@a0dcd9  
SimpleReceiver@a0dcd9 [8] :: Starting Execution  
SimpleReceiver@a0dcd9 [8] :: Polling Input Channels for Events  
[SimpleSender@14b7453 -> SimpleReceiver@a0dcd9] :: Initialised Processing  
[SimpleSender@14b7453 -> SimpleReceiver@a0dcd9] :: Removing Event: Frame@e86da0, with Time: 10  
SimpleReceiver@a0dcd9 [8] :: Event Found on Channel: Channel@b166b5  
[SimpleSender@1d5550d -> SimpleReceiver@a0dcd9] :: Initialised Processing  
[SimpleSender@1d5550d -> SimpleReceiver@a0dcd9] :: Removing Event: Frame@16a9d42, with Time: 10  
SimpleReceiver@a0dcd9 [8] :: Event Found on Channel: Channel@cdfc9c  
SimpleReceiver@a0dcd9 [8] :: Starting Event Execution  
SimpleReceiver@a0dcd9 [8] :: Received frame: -1614928625 with time: 10  
[SimpleSender@14b7453 -> SimpleReceiver@a0dcd9] :: Removing Event: Frame@ab95e6, with Time: 11  
SimpleReceiver@a0dcd9 [8] :: Received frame: 1066478705 with time: 10  
[SimpleSender@1d5550d -> SimpleReceiver@a0dcd9] :: Removing Event: Frame@1430b5c, with Time: 11  
SimpleReceiver@a0dcd9 [8] :: Received frame: 1483572524 with time: 11  
[SimpleSender@14b7453 -> SimpleReceiver@a0dcd9] :: Removing Event: Frame@cd2c3c, with Time: 12  
SimpleReceiver@a0dcd9 [8] :: Received frame: 1401655350 with time: 11  
[SimpleSender@1d5550d -> SimpleReceiver@a0dcd9] :: Removing Event: Frame@19616c7, with Time: 12
3. Scheduling Internal Critical Channel

Task@c21495 Task Started Execution
Task@c21495 Task Copying Startup ScheduledLPs, Found: 2
Task@c21495 Starting Execution of LP: SimpleSender@14b7453
SimpleSender@14b7453 [8] :: Starting Execution
SimpleSender@14b7453 [8] :: Polling Input Channels for Events
SimpleSender@14b7453 [8] :: Source LP Generating Events

[SimpleSender@14b7453 -> SimpleReceiver@a0dcd9] :: Adding Event: Frame@127734f, with Time: 2
[SimpleSender@14b7453 -> SimpleReceiver@a0dcd9] :: Adding Event: Frame@1546e25, with Time: 3
[SimpleSender@14b7453 -> SimpleReceiver@a0dcd9] :: Adding Event: Frame@a470b8, with Time: 4
SimpleSender@14b7453 [8] :: Frames Sent: 3
SimpleSender@14b7453 [8] :: Updating Clock To: 4

[SimpleSender@14b7453 -> SimpleReceiver@a0dcd9] :: Scheduling Destination LP Locally
SimpleReceiver@a0dcd9 [8] :: Now Scheduled Locally By: Sender1
SimpleSender@14b7453 [8] :: Source LP Recheduling Self Globally
SimpleSender@14b7453 [8] :: Now Scheduled Globally
Simulation Scheduling Task: Task@c21495
SimpleSender@14b7453 [8] :: Finished Execution

4. Scheduling External Critical Channel

Task@c21495 Task Started Execution
Task@c21495 Task Copying Startup ScheduledLPs, Found: 1
Task@c21495 Starting Execution of LP: SimpleSender@14b7453
SimpleSender@14b7453 [8] :: Starting Execution
SimpleSender@14b7453 [8] :: Initialising & Polling Input Channels for Events
SimpleSender@14b7453 [8] :: Source LP Generating Events

[SimpleSender@14b7453 -> SimpleReceiver@1d5550d] :: Adding Event: Frame@158b649, with Time: 4
[SimpleSender@14b7453 -> SimpleReceiver@1d5550d] :: Adding Event: Frame@1df073d, with Time: 5
SimpleSender@14b7453 [8] :: Event with time > simulation end time encountered, discarding.
SimpleSender@14b7453 [8] :: Frames Sent: 3
A Simulation Kernel

SimpleSender@14b7453 [8] :: Updating Clock To: 6
SimpleSender@14b7453 -> SimpleReceiver@1d5550d [8] :: Updating Clock To: 16
SimpleSender@14b7453 -> SimpleReceiver@1d5550d [8] :: Scheduling Destination LP Globally
SimpleReceiver@1d5550d [8] :: Now Scheduled Globally
Simulation Scheduling Task: Task@a0dcd9
SimpleSender@14b7453 [8] :: Source LP Recheduling Self Globally
SimpleSender@14b7453 [8] :: Now Scheduled Globally

5. Executing Events Up To The Safe Time

Task@c21495 Starting Execution of LP: SimpleReceiver@a0dcd9
SimpleReceiver@a0dcd9 [8] :: Starting Execution
SimpleReceiver@a0dcd9 [8] :: Initialising & Polling Input Channels for Events
SimpleSender@14b7453 -> SimpleReceiver@a0dcd9 [8] :: Initialised Processing
SimpleSender@14b7453 -> SimpleReceiver@a0dcd9 [8] :: Removing Event: Frame@e86da0, with Time: 11
SimpleReceiver@a0dcd9 [8] :: Event Found on Channel: Channel@1d99a4d
SimpleSender@1d5550d -> SimpleReceiver@a0dcd9 [8] :: Initialised Processing
SimpleSender@1d5550d -> SimpleReceiver@a0dcd9 [8] :: Removing Event: Frame@506411, with Time: 11
SimpleReceiver@a0dcd9 [8] :: Event Found on Channel: Channel@12152e6
SimpleSender@a0dcd9 [8] :: Starting Event Execution
SimpleReceiver@a0dcd9 [8] :: Received frame with time: 11
SimpleSender@14b7453 -> SimpleReceiver@a0dcd9 [8] :: Removing Event: Frame@1754ad2, with Time: 22
SimpleReceiver@a0dcd9 [8] :: Received frame with time: 11
SimpleReceiver@a0dcd9 [8] :: Empty Channel Encountered with Time: 11
SimpleReceiver@a0dcd9 [8] :: Finished Event Execution. Processed 2 Events. Final Safe Time: 11.
SimpleReceiver@a0dcd9 [8] :: Updating Clock To: 11
SimpleReceiver@a0dcd9 [8] :: Setting Critical: Channel@12152e6
SimpleSender@14b7453 -> SimpleReceiver@a0dcd9 [8] :: Terminated Processing
SimpleSender@1d5550d -> SimpleReceiver@a0dcd9 [8] :: Terminated Processing
SimpleReceiver@a0dcd9 [8] :: Finished Execution
Task@c21495 Task Finished Execution

6. Receiving Events From Channels

SimpleSender@14b7453 [8] :: Source LP Generating Events
SimpleSender@14b7453 -> SimpleReceiver@a0dcd9 [8] :: Adding Event: Frame@1fae3c6, with Time: 11
SimpleSender@14b7453 -> SimpleReceiver@a0dcd9 [8] :: Adding Event: Frame@7ffe01, with Time: 11
SimpleSender@14b7453 [8] :: Frames Sent: 2
... SimpleSender@1d5550d [8] :: Source LP Generating Events
SimpleSender@1d5550d -> SimpleReceiver@a0dcd9 [8] :: Adding Event: Frame@13582d, with Time: 1
SimpleSender@1d5550d [8] :: Frames Sent: 1
... SimpleReceiver@a0dcd9 [8] :: Starting Execution
A Simulation Kernel

SimpleReceiver@a0dcd9 [8] :: Initialising & Polling Input Channels for Events

[SimpleSender@14b7453 -> SimpleReceiver@a0dcd9] :: Initialised Processing
[SimpleSender@14b7453 -> SimpleReceiver@a0dcd9] :: Removing Event:
Frame@1fae3c6, with Time: 21

SimpleReceiver@a0dcd9 [8] :: Event Found on Channel: Channel@21b6d

SimpleReceiver@a0dcd9 [8] :: Initialising & Polling Input Channels for Events

SimpleSender@14b7453 [8] :: Source LP Generating Events

[SimpleSender@14b7453 -> SimpleReceiver@a0dcd9] :: Adding Event:
Frame@c9ba38, with Time: 12

SimpleReceiver@a0dcd9 [8] :: Initialising & Polling Input Channels for Events

[SimpleSender@14b7453 -> SimpleReceiver@a0dcd9] :: Initialised Processing
[SimpleSender@1d5550d -> SimpleReceiver@a0dcd9] :: Initialised Processing
[SimpleSender@1d5550d -> SimpleReceiver@a0dcd9] :: Removing Event:
Frame@13582d, with Time: 11

SimpleReceiver@a0dcd9 [8] :: Event Found on Channel: Channel@56a499

SimpleReceiver@a0dcd9 [8] :: Empty Channel Encountered with Time: 11

SimpleReceiver@a0dcd9 [8] :: Finished Event Execution. Processed 1 Events. Final Safe Time: 11.

SimpleReceiver@a0dcd9 [8] :: Updating Clock To: 11

... SimpleSender@a0dcd9 [8] :: Source LP Generating Events

[.SimpleSender@14b7453 -> SimpleReceiver@a0dcd9] :: Adding Event:
Frame@1e0be38, with Time: 12

SimpleReceiver@a0dcd9 [8] :: Frames Sent: 2

... SimpleReceiver@a0dcd9 [8] :: Starting Execution

SimpleReceiver@a0dcd9 [8] :: Initialising & Polling Input Channels for Events

[SimpleSender@14b7453 -> SimpleReceiver@a0dcd9] :: Initialised Processing
[SimpleSender@1d5550d -> SimpleReceiver@a0dcd9] :: Initialised Processing
[SimpleSender@1d5550d -> SimpleReceiver@a0dcd9] :: Removing Event:
Frame@16a9d42, with Time: 12

SimpleReceiver@a0dcd9 [8] :: Event Found on Channel: Channel@56a499

SimpleReceiver@a0dcd9 [8] :: Starting Event Execution

7. Discarding Events Past The Simulation End Time

Task@c21495 Task Started Execution
Task@c21495 Task Copying Startup ScheduledLPs, Found: 2
Task@c21495 Starting Execution of LP: SimpleSender@14b7453

SimpleSender@14b7453 [8] :: Starting Execution
SimpleSender@14b7453 [8] :: Initialising & Polling Input Channels for Events
SimpleSender@14b7453 [8] :: Source LP Generating Events
SimpleSender@14b7453 [8] :: Event with time > simulation end time encountered, discarding.

SimpleSender@14b7453 [8] :: Frames Sent: 1
SimpleSender@14b7453 [8] :: Starting Event Execution
SimpleSender@14b7453 [8] :: Finished Event Execution. Processed 0 Events. Final Safe Time: 1.

SimpleSender@14b7453 [8] :: Updating Clock To: 1

[SimpleSender@14b7453 -> SimpleReceiver@a0dcd9] :: Updating Clock To: 11
SimpleSender@14b7453 [8] :: Source LP Recheduling Self Globally
SimpleSender@14b7453 [8] :: Finished Execution
8. SELF SCHEDULED EVENTS

Task@c21495 Task Started Execution
Task@c21495 Task Copying Startup ScheduledLPs, Found: 1
Task@c21495 Starting Execution of LP: SimpleSender@14b7453
SimpleSender@14b7453 [8] :: Starting Execution
SimpleSender@14b7453 [8] :: Initialising & Polling Input Channels for Events
SimpleSender@14b7453 [8] :: Source LP Generating Events
[SimpleSender@14b7453 -> SimpleReceiver@1d5550d] :: Adding Event:
  Frame@12152e6, with Time: 2
SimpleSender@14b7453 [8] :: Frames Sent: 1
SimpleSender@14b7453 [8] :: Updating Clock To: 2
...
SimpleReceiver@1d5550d [8] :: Initialising & Polling Input Channels for Events
[SimpleSender@14b7453 -> SimpleReceiver@1d5550d] :: Initialised Processing
[SimpleSender@14b7453 -> SimpleReceiver@1d5550d] :: Removing Event:
  Frame@12152e6, with Time: 12
SimpleReceiver@1d5550d [8] :: Event Found on Channel: Channel@e86da0
SimpleReceiver@1d5550d [8] :: Starting Event Execution
SimpleReceiver@1d5550d [8] :: Received frame with time: 12. Resending to self.
SimpleReceiver@1d5550d [8] :: Self Event Scheduled: Frame@1f9dc36 with Time: 13
SimpleReceiver@1d5550d [8] :: Empty Channel Encountered with Time: 12
SimpleReceiver@1d5550d [8] :: Received frame with time: 12. Resending to self.
SimpleReceiver@1d5550d [8] :: Self Event Scheduled: Frame@12152e6 with Time: 13
SimpleReceiver@1d5550d [8] :: Empty Channel Encountered with Time: 12
SimpleReceiver@1d5550d [8] :: Updating Clock To: 12
SimpleReceiver@1d5550d [8] :: Setting Critical: Channel@e86da0
[SimpleSender@14b7453 -> SimpleReceiver@1d5550d] :: Terminated Processing
SimpleReceiver@1d5550d [8] :: Finished Execution
Task@c21495 Task Finished Execution
Appendix B: RESULTS TABLES

1. SIMULATION SCALE RESULTS

SMALL REQUEST RESPONSE MODEL RESULTS

<table>
<thead>
<tr>
<th>Processor Count</th>
<th>Time (seconds)</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>72.65</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>74.22</td>
<td>0.9788</td>
</tr>
<tr>
<td>3</td>
<td>77.81</td>
<td>0.9336</td>
</tr>
<tr>
<td>4</td>
<td>79.69</td>
<td>0.9116</td>
</tr>
</tbody>
</table>

Table B.1 Small Request Response Model Results

MEDIUM REQUEST RESPONSE MODEL RESULTS

<table>
<thead>
<tr>
<th>Processor Count</th>
<th>Time (seconds)</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>72.65</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>74.22</td>
<td>0.9788</td>
</tr>
<tr>
<td>3</td>
<td>77.81</td>
<td>0.9336</td>
</tr>
<tr>
<td>4</td>
<td>79.69</td>
<td>0.9116</td>
</tr>
</tbody>
</table>

Table B.2 Medium Request Response Model Results

LARGE REQUEST RESPONSE MODEL RESULTS

<table>
<thead>
<tr>
<th>Processor Count</th>
<th>Time (seconds)</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>68.86</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>57.4</td>
<td>1.1996</td>
</tr>
<tr>
<td>3</td>
<td>53.1</td>
<td>1.2967</td>
</tr>
<tr>
<td>4</td>
<td>49.7</td>
<td>1.3855</td>
</tr>
</tbody>
</table>

Table B.3 Large request response model results.

2. EVENT GRANULARITY RESULTS

VERY LOW GRANULARITY EVENTS

<table>
<thead>
<tr>
<th>Processor Count</th>
<th>Time (seconds)</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>56.73</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>45.93</td>
<td>1.2351</td>
</tr>
</tbody>
</table>
A Simulation Kernel

Table B.4 Very low event granularity test results.

<table>
<thead>
<tr>
<th>Processor Count</th>
<th>Time (seconds)</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>61.44</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>42.6</td>
<td>1.4422</td>
</tr>
<tr>
<td>3</td>
<td>35.4</td>
<td>1.7355</td>
</tr>
<tr>
<td>4</td>
<td>33.9</td>
<td>1.8123</td>
</tr>
</tbody>
</table>

Table B.5 Low event granularity test results.

<table>
<thead>
<tr>
<th>Processor Count</th>
<th>Time (seconds)</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>64.98</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>41.85</td>
<td>1.5526</td>
</tr>
<tr>
<td>3</td>
<td>33.04</td>
<td>1.9664</td>
</tr>
<tr>
<td>4</td>
<td>29.56</td>
<td>2.1978</td>
</tr>
</tbody>
</table>

Table B.6 High event granularity test results.

<table>
<thead>
<tr>
<th>Processor Count</th>
<th>Time (seconds)</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>68.2</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>37.78</td>
<td>1.8051</td>
</tr>
<tr>
<td>3</td>
<td>26.68</td>
<td>2.5562</td>
</tr>
<tr>
<td>4</td>
<td>22.835</td>
<td>2.9866</td>
</tr>
</tbody>
</table>

Table B.7 Very high event granularity test results.

3. **Time Parallelism Results**

<table>
<thead>
<tr>
<th>Processor Count</th>
<th>Time (seconds)</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>61.52</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>34.24</td>
<td>1.7967</td>
</tr>
<tr>
<td>3</td>
<td>27.5</td>
<td>2.2370</td>
</tr>
<tr>
<td>4</td>
<td>25.9</td>
<td>2.3752</td>
</tr>
</tbody>
</table>
Appendix C: Code Snippets

1. Event Execution Loop

```java
// look at the first event in the queue to be executed
Event executingEvent = getEventQueue().peek();

// while there are events waiting and valid to be executed on this LP
while (executingEvent != null && executingEvent.getTime() <= this.localWindow) {
    getEventQueue().remove();
    // we need to get the originating channel before we process the event
    // otherwise if the event gets sent on another channel during the processing
    // it will create problems
    eventChannel = executingEvent.getChannel();
    eventChannel = executingEvent.getChannel();
    // process the event, this method is implemented by a subclass
    processEvent(executingEvent);
    // if the event was not self scheduled
    if (eventChannel != null) {
        nextEvent = eventChannel.getNextEvent();
        // if there are more events in the channel
        if (nextEvent != null) {
            this.getEventQueue().add(nextEvent);
        } else {
            eventChannel.setSampled(false);
            updateLocalWindow(eventChannel);
        }
    }
    // get the next event
    executingEvent = getEventQueue().peek();
}
```

2. Event Execution Loop

```java
// while LPs are still scheduled in this task
while (!this.getThreadScheduledLPs().isEmpty()) {
    // get the next scheduled LP
    executingLP = this.getNextScheduledLP();
    // make sure this is the only thread executing the LP
    synchronized (executingLP) {
        this.executingLP = executingLP;
    }
```
 Appendix D: RUNNING INSTRUCTIONS

The simulation kernel is executed via the command line and requires that the Java Runtime Environment (JRE) version 6+ is installed. The program can be run by executing the following command:

```
java SimulationKernel <model name> <processor count> <simulation end time> <source LP running time>
```

If a model other than SimplePipe or RequestResponse is required, such as a user developed model, then the Main.java file will require editing and recompiling. You will need to create an additional if then statement and include the name and path of your simulation model. You will also need to tell the Java compiler where to find your simulation model classes during compilation.

If an error is encountered trying to run the simulation then the error message should provide some guidance in how to fix it. If the problem persists please contact the author.