An Investigation Into the Application of Parallel Computers for the Dynamic Simulation of Chemical Processes

Roderick C. McKinnel

Doctor of Philosophy
University of Edinburgh
1994
Declaration

The work described in this thesis is the original work of the author and was carried out without the assistance of others, except where explicit credit is given in the text. It has not been submitted, in whole or in part, for any other degree at any University.
Acknowledgements

I would especially like to thank Professor Jack Ponton for initiating and supporting this project, Dr Eric Fraga for his help with the mathematics and proof reading, Neil Skilling for keeping the computer system up and running and Murray Laing for his infinite wisdom on Postscript.

Finally I would like to thank all those people in and outside of the department who have helped and supported me along the way.
Abstract

The detailed dynamic simulation of chemical processes is computationally expensive. Standard single processor (sequential) computers are not of sufficient power to tackle such simulations in a reasonable time frame. In particular, it is not possible to run complex simulations in less than real time. The solution to obtaining the processing power required lies in moving towards the use of multiple processor (parallel) computers. Unfortunately, obtaining the full benefit from parallelism requires the problem being solved to be partitionable into parts, each of which can be solved concurrently. For the majority of problems, locating this parallelism is not trivial.

An investigation into the use of MIMD parallel computers for dynamic process simulation has been performed. Initially the parallel dynamic simulation of distillation was studied. Later work moved on to the parallel dynamic simulation of complete processes. As a result, two parallel process simulators have been produced: PDist (Parallel Distillation simulator) and PNet (Parallel Process Network simulator). Throughout the work a parallel modular approach, rather than a parallel equation based approach, has been adopted. Results shown that the parallel modular approach maps efficiently to parallelism and that excellent reductions in execution time can be obtained.

As well as the exploitation of parallelism for processing power reasons, a large amount of the work aimed to show the benefits which the parallel modular approach offered from a usability point of view. Both PDist and PNet were designed with usability in mind. The simulation model interfaces created were designed to hide the majority of the parallelisation from the modeller. A large amount of work was also carried out on simulation input, interaction and graphical output. PDist and PNet are now much more than just concept provers. PDist is particularly usable and has been used for an industrial case study.
# Contents

## 1 Introduction

1.1 Parallel Processing ........................................... 3
   1.1.1 Development of the Technology ......................... 3
   1.1.2 Parallel Architecture Types ............................ 5
   1.1.3 Programming the New Architectures .................... 9

1.2 Computer Use in Chemical Engineering .......................... 11
   1.2.1 Current Computer Usage in Chemical Engineering ....... 12
   1.2.2 Requirements for the Future ......................... 13

1.3 Aims of Work ................................................. 14
   1.3.1 The Dynamic Simulation of Distillation ............... 16
   1.3.2 Complete Plant Simulation ............................. 17
   1.3.3 Interaction and Usability of Parallelism ............. 17

1.4 Summary .................................................. 18

## 2 Literature Review ........................................ 19

2.1 Introduction ............................................. 19
2.2 General Overviews ....................................... 20
2.3 General Equation Solving ............................... 23
  2.3.1 Linear Algebraic Equations ........................ 24
  2.3.2 Other Equation Types ................................ 31
2.4 Process Simulation ........................................ 33
   2.4.1 Steady State Simulation (Flowsheeting) .......... 34
   2.4.2 Dynamic Simulation ................................. 38
2.5 Process Synthesis ......................................... 40
2.6 Summary .................................................. 42

3 Dynamic Process Simulation ............................... 43
  3.1 Dynamic Simulation: Theory and Parallelism ........... 44
   3.1.1 Equation-Based Methods ............................ 46
   3.1.2 Modular Based Methods ............................. 50
   3.1.3 Comparison of Methods ............................. 54
  3.2 Parallel Dynamic Simulation of Distillation .......... 56
   3.2.1 Distillation ........................................ 57
   3.2.2 Modularising Distillation for Parallelism ........ 58
   3.2.3 Modular Algorithms Used ........................... 64
  3.3 Distillation Models Overview ........................... 66
   3.3.1 Model Overview .................................... 66
  3.4 Summary ................................................ 71
4 PDist

4.1 Introduction ........................................... 72

4.2 PDist Development History .............................. 75

4.3 PDist Software and Portability .......................... 78

4.4 Parallel Distillation Implementation ................... 80

4.4.1 Decomposition Overview ............................. 80

4.4.2 PDist Model Interface ............................... 83

4.4.3 Simulation Critical Communications .................. 87

4.4.4 Utility Communications ............................... 99

4.4.5 Complete Parallel Algorithm ........................ 102

4.4.6 Implementation Summary ............................. 106

4.5 The Front End .......................................... 106

4.5.1 The Dynamic Simulation Input Problem ............... 106

4.5.2 PDist Input Format .................................. 112

4.5.3 Accessing Input Information from PDist Models ...... 116

4.5.4 Interaction .......................................... 117

4.5.5 Front End Conclusion ............................... 117

4.6 The Back End .......................................... 118

4.6.1 Introduction ....................................... 119

4.6.2 Simulation Solution Management in PDist ............ 120

4.6.3 PDist's Graphical Tools ............................ 125
4.6.4 Back End Summary ........................................ 131
4.7 Summary ................................................... 132

5 PNet .......................................................... 134

5.1 Introduction ................................................. 134
  5.1.1 Connecting Dynamic Simulators ....................... 136
  5.1.2 PNet Approach to Simulator Construction .......... 140
  5.1.3 Other Simulator Connection Issues ................. 142
  5.1.4 The Current and Proposed Structure of PNet ....... 145
  5.1.5 Summary ............................................. 147

5.2 Creating a PNet Pipe ....................................... 148
  5.2.1 The PNet Pipe ....................................... 148
  5.2.2 Pipe Creation and the Pipe Interface Routines .... 151
  5.2.3 Example Pipes Created ................................ 156
  5.2.4 Summary ............................................. 160

5.3 Creating a PNet Connectable Simulator .................. 160
  5.3.1 The PNet Interface Routines ......................... 160

5.4 Describing and Executing a PNet Simulation ............. 165
  5.4.1 Describing a PNet Process ......................... 165
  5.4.2 Supporting Parallel Simulators .................... 168
  5.4.3 Execution of PNet ................................. 169
  5.4.4 Solution Display .................................... 172
5.5 Summary ......................................... 173

6 Evaluation ........................................ 174

6.1 PDist Results .................................... 174

6.1.1 The Usefulness of PDist ..................... 175
6.1.2 The Robustness of PDist ..................... 176
6.1.3 The Usability of PDist ....................... 178
6.1.4 Gain from Parallelisation ................... 179
6.1.5 Summary ................................... 189

6.2 PNet Results ................................... 189

6.2.1 Steady State Simulator Construction Results .... 190
6.2.2 Parallel Dynamic Simulation Results ........... 193
6.2.3 Summary ................................... 197

7 Conclusions and Future Work .................... 199

7.1 The Modular Approach .......................... 199
7.2 Load Balancing ................................ 200
7.3 Robustness .................................... 202
7.4 Flexibility .................................... 202
7.5 Usability ..................................... 203
7.6 User Interfaces ................................ 204
7.7 Summary ..................................... 205
CONTENTS

7.8 Future Work ........................................... 205
7.9 The Future ........................................... 207

A PDist Test Models ........................................... 209

A.1 General Tray Model Description ....................... 209
A.2 Tray Liquid and Vapour Flowrate Models ............... 213
A.3 Tray Hydraulic Models .................................. 214
A.4 Conventional Reflux Model Description ............... 217
A.5 Conventional Reflux Flowrate Models ............... 220
A.6 Liquid/Liquid Separator Reflux Model ............... 221
A.7 Liquid/Liquid Separator Flowrate Models ............. 224
A.8 Liquid/Liquid Isothermal Flash Model ............... 225
A.9 Conventional Reboiler Model Description .............. 228
A.10 Conventional Reboiler Flowrate Models ............... 230
A.11 Reactor/Reboiler Model Description ............... 231
A.12 Reactor/Reboiler Flowrate Models .................. 234
A.13 Vapour/Liquid Equilibrium Models ............... 236
A.14 Proportional/Integral Control Model ............... 238
A.15 Level Control Models ................................ 239
A.16 Simulation Input Ramping Models ............... 240
A.17 Simulation Input Oscillation Models ............... 241
A.18 The Regula-Falsi Method ................................ 242
D.2.3 Example PDist Loader Text under PNet ............ 284

E Input File Syntax Diagrams 288

E.1 PDist Graphics Description Syntax ................. 289
E.2 PDist Model Input Format .......................... 296
E.3 PNet Input File Syntax ........................... 304

F Bibliography 308
# List of Figures

1.1 CM-200 Fortran Example ........................................ 7

3.1 Modular Approach without Recycle ............................. 52

3.2 Modular Approach with Recycle ................................ 52

3.3 Single Distillation Vapour/Liquid Equilibrium Stage ........... 59

3.4 Information Flow for Various Distillation Solution Methods .... 60

3.5 Conventional Distillation Column ............................... 68

3.6 Reactive/Azeotropic Distillation Column ....................... 69

4.1 PDist Program Connection Strategy ............................. 81

4.2 PDist Model Interface ........................................... 83

4.3 Effect of Various Factors on Communication Efficiency ........ 89

4.4 Effect of Various Factors on Communication Efficiency ........ 90

4.5 Possible Communication and Control Configurations for PDist .. 93

4.6 Example of Simulation Input Format Functionality ............. 108

4.7 The Interaction Panel ............................................ 118

4.8 Example of Solution Structure Declaration Format ............. 122
4.9 Example of Data Management Recommendations for 3D Graphs 125
4.10 PDist Graphical Display Selection Panel .......................... 127
4.11 Selection of 3D Context Items ...................................... 127
4.12 Example of Solution 3D Graphical Window ...................... 129
4.13 Example of Solution 2D Graphical Window ...................... 130
4.14 Example of Solution Tabulation Mechanism ..................... 131
4.15 Example of Run Time Graphics ..................................... 132

5.1 Diagram of General PNet Structure ................................. 135
5.2 The PNet Connection Manager: The Pipe .......................... 140
5.3 Diagram Showing the Implementation Structure of PNet ........ 146
5.4 Structure of the Pipe Interface Routines .......................... 152
5.5 Simple Dynamic Mixer Using PNet ................................ 164
5.6 Example PNet Process Description ................................. 167
5.7 Example Input Description with System Call ..................... 170
5.8 Example PNet Loader Command ................................... 171
5.9 Screen Dump of the PNet Solutions Viewer ....................... 172

6.1 Timings Plot for Simulation With Simple Physical Properties ... 183
6.2 Timings Plot for Simulation With Complex Physical Properties 184
6.3 Parallel Efficiency Plot for Simple and Complex Simulations ... 185
6.4 Graph of Calculation Load Distribution During 10 Processor Simula-  

LIST OF FIGURES

6.5 Graph of Communications Load Distribution During 10 Processor Simulation .............................................. 187

6.6 Process Flowsheet for PDist Column Sequence .......................... 194

6.7 Process Flowsheet for PDist Column Sequence with Recycle .... 195

A.1 Distillation Tray $n$ in a Stage Section .......................... 210

A.2 Figure Showing Hydraulic Features of a Distillation Tray .... 215

A.3 Conventional Reflux Section .......................... 217

A.4 Reactive Distillation Reflux Section .......................... 221

A.5 Isothermal Flash ............................................. 225

A.6 Conventional Reboiler Section .......................... 228

A.7 Reactor/Reboiler Section ............................................. 232

A.8 Oscillation Approach Used ............................................. 241

A.9 The Regula-Falsi Method ............................................. 243
List of Tables

1.1 Flynn’s Taxonomy .................................................. 5

2.1 Table of Applications for Parallelism in Chemical Engineering .. 21
Chapter 1

Introduction

Since the development of the computer in the late 1940's, engineers have been able to tackle ever more computationally intensive problems. Unfortunately the computational requirements always exceed that which the current computer hardware can provide. As new computational speeds are achieved, users either find new problems to tackle which were previously thought too difficult, or remove some simplifications from their existing models. Either way there is a self perpetuating loop of computational requirements to hardware requirements back to computational requirements.

Until the early 1970's all computers were based on the so called von Neumann architecture. This architecture incorporates one central processor, connected to one central memory, executing one instruction at a time. Computers with this type of architecture are more commonly referred to as “sequential”. Sequential computers make up the majority of computers in use today. Great advances in chip, memory and circuit board technology have meant that these computers are still growing in computational performance and are likely to for some years to come. In conjunction with this scientists have been developing specialised algorithms, geared specifically to obtain the maximum performance from the sequential architecture.
Unfortunately there is a physical upper limit to the computational speed obtainable with sequential machines. For engineers requiring computational performance more than two orders of magnitude greater than current levels, not only improved computer components will be required, but a different architecture which will be scalable enough to provide the performance engineers are demanding for the foreseeable future.

It is now widely recognised that the solution to future computing requirements lies in moving from sequential to "parallel" architectures. Parallel architectures differ in that they allow the execution of two or more sets of computer instructions at the same time. This is performed using specialised numerical hardware or multiple, usually von Neumann style, processors. Computers incorporating parallel concepts, or supercomputers, began to appear in the early 1970's. Until now they have tended to be mainly research tools, with the majority of machines being installed in research establishments. However the 1990's has seen an ever widening range of relatively low cost/performance machines beginning to appear. With the lowering costs and better software being provided by vendors, parallel machines are slowly beginning to make a place for themselves in the industrial arena.

The main problem with these new parallel architectures is that the extra performance comes at a cost. Problems must be partitionable into separate and concurrently executable pieces. The more concurrency obtained, the better the performance. This means that existing algorithms must be rewritten to suit the new architectures. This is not a simple task. Unlike with sequential architectures, there are many types of parallel architecture. Each of these require different programming strategies to produce the maximum performance. Each architecture also exploits parallelism at different levels, or granularities. The granularity goes from fine to coarse grained. The coarser the grain the larger the concurrent tasks become. Most problems are best solved using a specific granularity. This
somewhat limits the range of architectures that can be used. It also presents a problem when two or more coupled problems, or a given solution method, require different granularities to be solved. In this case a compromise must be made or a heterogeneous approach to problem solving employed.

1.1 Parallel Processing

This section gives a brief overview of the development of parallel computers and the various architecture types. Included in the description are some of the main "buzz" words and acronyms used throughout this thesis. Most of the information regarding the various different machine types was obtained from Trew and Wilson [1].

1.1.1 Development of the Technology

In the mid 1940s John von Neumann, who originally trained as a Chemical Engineer [2], proposed the first basic design for the computer. The design involved having a single processing unit connected to a single store of memory. The processor fetched instructions and operands from the memory, performed a computation and wrote the results back in to memory. The first computers produced of this type were composed of vacuum tubes and tended to be unreliable. Even at this early stage, von Neumann realised the potential of parallelism but could not put it in to practice with the current hardware available.

The 1960's saw the vacuum tube replaced by solid state components. The first minicomputers and mainframes began to appear. The operating systems on these machines incorporated inter process communication, time-sharing and memory management and were designed to be used by all, from small businesses to large companies.
In 1976 the first ever "vector supercomputer" appeared on the market. This was produced by Cray Research Incorporated and was the brain child of Seymour Cray. The Cray 1 was based on Control Data Corporations CDC7600 and was the first computer to incorporate pipelined vector processing hardware. This hardware allowed independent tasks to be executed concurrently in a similar manner to a production line. Since the Cray 1 many different Crays have been produced and Cray Research Inc still produce some of the fastest computers available today.

By the end of the 1970s the idea of parallel computing was beginning to take off. The development of Very Large System Integration (VLSI) technology was allowing hundreds of thousands of transistors on a single chip. Programming was being made easier by higher level languages such as Fortran, Pascal and C. At the same time many techniques for handling concurrency had been developed such as Semaphores, Monitors and Signals and were being taught widely to computer science students giving rise to a large number of well versed programmers.

The first multiprocessor computer produced was the Illiac IV. This was designed at Carnegie-Mellon University. The machine was based on many new ideas but failed due to lack of usable programming languages and environments and its reliance on very state of the art hardware. Vendors learned many lessons from the failure of this machine and, as a result, most modern computers are build using very common processing units.

Throughout the 1980s, the use of parallel processing increased greatly. This was mainly brought about by the lowering cost of the parallel hardware. With a large number of parallel machines being installed in research establishments, the software environments were also beginning to improve.

Now in the 1990s, parallel hardware is ever cheaper. The software is slowly catching up with developments and with most hardware now being hosted by
Unix workstations, the mechanism for creating and running parallel programs is very similar to that used for sequential ones. The technology is now at a state where it is ready to move from academia into industry. The main difficulty is in locating where parallelism can be applied and targeting the appropriate parallel architecture.

### 1.1.2 Parallel Architecture Types

Parallel architectures are usually classified by their general approach to parallelism. In 1967 Michael J. Flynn [3] put forward his classification taxonomy of computer architectures. Table 1.1 below shows the different classifications.

<table>
<thead>
<tr>
<th>Single Data Stream</th>
<th>Single Instruction Stream</th>
<th>Multiple Instruction Stream</th>
</tr>
</thead>
<tbody>
<tr>
<td>SISD (von Neumann, PC ..)</td>
<td>MISO</td>
<td></td>
</tr>
<tr>
<td>SIMD (DAP, CM, MP-1)</td>
<td>MIMD (Cray, CM5, Meiko ..)</td>
<td></td>
</tr>
</tbody>
</table>

Table 1.1: Flynn’s Taxonomy

The classifications divide architectures into groups based on how instructions are executed and on the data structure which they operate. An Instruction Stream is “the sequence of instructions as performed by the machine”. A Data Stream is “the sequence of data called for by the instruction stream (including input and partial or temporary results)”. The different classifications are now described:

**SISD: Single Instruction Stream-Single Data Stream**

SISD is the general classification into which the von Neumann style computer architecture fits. It describes a computer which executes one instruction at a time operating on one data stream. Current sequential computers are classified as SISD although most modern machines are slight divergences from the classic von Neumann architecture.

**SIMD: Single Instruction Stream-Multiple Data Stream**
This classification refers to an architecture type in which many processors execute the same instruction but on different data. Such machines are often referred to as "data parallel", "massively parallel" or "fine grained" machines. The best known machines of this type are the Connection Machine (CM), the AMT Digital Array Processor (AMT DAP) and the MasPar MP-1.

All these machines use large numbers of simple processors: up to 65,536 in the CM. Data is distributed between the processors and instructions are broadcast to each processor from a central program. The machines work in "lock step". This involves every processor receiving the same instruction. At the end of each instruction the processors synchronise and then move on to the next instruction. The "lock step" process makes the parallelism simpler to control and helps to avoid dead lock problems for communications. Interprocessor communication is available and performed using vast connection arrays. The machines are particularly good for problems where processor communication is nearest neighbour.

SIMD machines tend to be suited to problems where parallelism can be found at a very fine grain. This can be at a physical or algorithmic level. They are especially suited to problems such as image processing where pixel calculations can be distributed and mesh problems such as finite difference calculations.

To program these machines a number of current high level languages have been extended to include array calculations. A simple fortran example for the CM-200 [4] is shown in figure 1.1 below.

The example shows the general way SIMD machines are programmed. The nature of the machines means that they are most efficient when matrices are large and all of the processors can be used. For more complex operations such as matrix multiplication and QR factorisation, specialised routines are typically provided as writing routines to perform such operations requires
Conventional Fortran 77

\begin{verbatim}
double precision A(100)
!
add 1 to every element of A
do i=1,100
   A(I) = A(I) + 1
end do
\end{verbatim}

CM-200 Fortran Extension

\begin{verbatim}
double precision A(100)
!
add 1 to every element of A
A = A + 1
\end{verbatim}

Figure 1.1: CM-200 Fortran Example

a good knowledge of the hardware to get the problem decomposition right and thus the best performance. Similar routines are also available for vector processor machines like the Cray series.

The programming style for these machines is probably their best feature. Given a standard language it should be possible to port code across different machine types. Unfortunately most vendors supply their own language extensions and there is no sign of standardisation at present above and beyond existing high level languages. The main drive by some vendors seems to be towards supporting the new Fortran 90 style syntax. This allows many of the matrix operations required by SIMD machines to be declared in a meaningful and usable manner. Until recently there were no Fortran 90 compilers available.

Overall SIMD machines make up a substantial amount of current supercomputer usage along with vector processors such as the Cray series. The SIMD approach works well for many problems. Unfortunately the approach is not applicable to all problems and not realistically scalable enough to provide future performance requirements. There is now a move by vendors towards coarser grained parallelism using fewer but much more powerful processors. Some of these new machines are capable of being run in a SIMD manner but are designed for more coarse grained parallelism. These machines belong
to the classification MIMD and are explained below.

MISD: Multiple Instruction Stream-Single Data Stream

There are currently no computers of this classification in existence. It is difficult to visualise how such a machine could function and whether any great benefit could be obtained from such an architecture.

MIMD: Multiple Instruction Stream-Multiple Data Stream

The MIMD classification of architectures covers any architecture in which the processors can run different instructions concurrently and where each processor has access to different data streams.

MIMD is a natural progression from SISD and is regarded as the way forward for parallel computing. The MIMD approach involves running complete programs on different processors, where each processor is usually of equal power. These programs are stored locally by the processor and communicate to other programs by way of hardware/software communications links. The programs are often referred to as “Communicating Sequential Processes” and most communication software attempts to emulate the communication theory described by Hoare [5].

MIMD machines tend to be built from either small numbers of powerful processors or larger numbers of smaller processors. In the machines with smaller numbers of processors the memory used is quite often “shared memory”. In these machines each processor can either access all of the memory or in certain cases private blocks of the memory. An example of a MIMD shared memory machine is the Cray Y-MP. This uses a number of very powerful vector processors connected to a central memory store. The main advantage of shared memory is that they are generally simpler to program. Unfortunately as the number of processors increases the contention for memory usage becomes significant and a bottleneck occurs. For this reason shared memory machines are somewhat limited in terms of future
development.

The other form of memory usage is "distributed memory". In machines with distributed memory each processor has its own memory store. Programming these machines is more difficult since information must be explicitly communicated between programs due to lack of direct memory access. Some machines such as the BBN Butterfly use distributed memory, but by way of a communications network library mimic shared memory usage.

Distributed memory machines offer the greatest performance since they are technically very scalable. Companies such as Thinking Machines, who manufacture the Connection Machine series, have now turned to this type of architecture. Their main intention is to be the first company to produce a TeraFLOPS machine. The new Connection Machine, the CM5, is a 1000 processor MIMD machine. Unlike the earlier SIMD CM-200 this machine has much fewer but more powerful processors and is designed to support the MIMD style of architecture. As a selling point it is also capable of running SIMD style code from previous machines. Machines of this type represent the future of computing at the high performance end of the market.

1.1.3 Programming the New Architectures

The biggest drawback to parallel processing at present is the lack of standard software environments and tools. Almost all vendors supply their own compilers which either extend the functionality of existing languages such as Fortran or C, or provide specialised libraries which allow the user to perform specialised hardware tasks.

The main requirement is for a language which allows the parallel operations of SISD, SIMD and MIMD machines to be expressed in a simple and meaningful manner. Unfortunately there is always resistance to change and most people
want to stick with their existing languages. Also vendors are not going to use languages developed by their competitors. This is not to say that nobody has attempted to produce a parallel language. Probably the most popular example is OCCAM [6]. This was specifically developed as the programming language for the INMOS Transputer. In the words of Fountain [6], "Occam is the first language to be based upon the concept of parallel, in addition to sequential execution, and to provide automatic communication and synchronisation between concurrent processes". When transputer based systems first appeared OCCAM was used by almost everybody. Unfortunately OCCAM lacked many of the features provided by most high level languages and was conceptually difficult to program. Its other main drawback was that it had to be used from within a development system. This was keyboard based and involved many alien style tools such as a folding editor. There are still a lot of people using OCCAM today, but this is reducing rapidly with most people referring back to extended versions of classic high level languages. Programs written in existing high level languages are more easily ported to other architectures since most processors have Fortran and C compilers. OCCAM does not appear to be dead however. INMOS's new processor the H1 (or T9000) is due out in early 1994. Along with this a revised OCCAM is going to be implemented. This has extensions to provide it with more of the functionality of high level languages. It is also going to be usable under the X-Window system and ported on to a wide range of computer architectures. At some point languages are going to have to standardise and OCCAM has played and is likely to play a leading role in researching exactly what future languages should be like.

It has already been mentioned that some vendors provide specialised libraries for performing mathematical tasks, especially for vector and SIMD architectures. In an attempt to provide a more standard set of mathematical libraries the Basic Linear Algebra Subroutines (BLAS) [7,8] have been developed. These are designed to provide a set of standard routines to perform basic linear algebra operations. They are written in three levels: Level-1, Level-2 and Level-3. Levels
2 and 3 are developments from level 1 [9,10]. Level-2 BLAS was very successful at exploiting parallelism in vector machines, but not so good for coarse grained machines and SIMD machines. Level-3 BLAS takes into account these other machines by including matrix/matrix operations and has proved very successful. From these basic routines it is possible to build systems to solve most numerical problems. These have been included in such software packages as LINPACK which is now used for benchmark comparisons of various machines. The BLAS routines are available on vector machines such as the Cray series and are implemented to some extent on SIMD machines such as the CM. There is talk of producing similar routines for the NAG library for MIMD type architectures, but as yet it is mostly speculation [11].

The programming of MIMD systems still requires specialised tools. This is likely to continue for some years until either standard communication software extensions to existing languages or a good and generally acceptable parallel language is produced. The main software challenge for the application writer at present is to locate the parallelism and program it as well as possible given current tools.

1.2 Computer Use in Chemical Engineering

Chemical engineering is a very numerical subject and spans a wide range of problem types. For this reason computers have become well established as an integral part of everyday engineering for a great many years. Many of the typical chemical engineering problems can now be solved quite efficiently on modern computers. For these problems there is little need for larger machines. However there is a still a set of extremely computationally intensive problems for which there is currently no quick solution.
1.2.1 Current Computer Usage in Chemical Engineering

There now exists a wide range of software to perform many common chemical engineering tasks. Typical everyday tasks performed are:

- Modelling and Design/Synthesis
- Flowsheeting
- Control Simulations
- Thermodynamics and Physical Property Estimation
- Simple Equation Solving (P.D.Es, O.D.Es, Non-linear equations etc.)
- Optimisation
- Word Processing, Spread Sheeting etc.

Most of this software requires more user time than computer time, with packages running reasonably quickly on current machines. From the industrial viewpoint current software is probably sufficient enough to let them function to a reasonable level. For this reason there is a high degree of resistance to change in terms of computer usage. Most companies still adopt a predominantly PC/VAX based computer system with limited networking. In these areas modern computing is going to take some time to break through since there is not enough incentive for companies to change. In the end the main incentive is going to come from the future requirements for processing power. These requirements are being driven by external influences beyond the control of single companies.
1.2.2 Requirements for the Future

The world is rapidly becoming more constrained with low emission limits and energy conservation measures being enforced. To make sure that chemical plants meet the new standards, much more optimisation, simulation and analysis of alternatives is required. A lot of research has gone into designing suitable algorithms and approaches for tackling these problems. A reasonable amount of software has also been written to implement these new algorithms. The main problem arises when the programs begin to include the more complex attributes required to provide truly realistic solutions. Simply adding heat integration considerations to a distillation train synthesis program can give rise to a program which takes up to 24 hours to run on a modern workstation. The growth in computation required is exponential for these problems and for the engineer wishing to get a result in a few minutes they are either going to have to put up with a simplified solution or buy a computer which has the processing power they require.

With current sequential computers there is a maximum obtainable performance. This is dictated by the speed of light and the physical limit to circuit densities on a chip. To get the performance required by major applications, multiple processors will be required. The current thinking of many people is that processors will keep getting faster and produce the performance required. Even if this were the case it is quite often cheaper to use multiple old processors to get the performance than the current state of the art chip technology. Also, the world is demanding results now and is not going to wait decades for single processor computers to get fast and cheap enough to use. The technology to get the power required exists now in the form of parallelism. If parallelism is targeted for use now, even when processor speeds increase, the multiple processor approach should always provide much better performance than the single processor approach when compared on equal terms.
To design and operate the plants of the future, the engineer is also going to require a multitude of simple and specialised tools. This will involve integrating existing programs running on current machines with those specialised parallel programs which require specialised hardware. With engineer's time becoming ever more expensive, new packages are required which aid them to do their work more efficiently rather than hinder them. As packages tackle ever more difficult problems so the number of parameters for these will increase. Somehow most of these will have to be managed for the user. Graphical User Interfaces (GUI), are required which allow the user to move and manipulate information in a standard way within and between different programs. This is especially true for parallel programs where there is the added complexity of multiple program execution. Technology has now reached a state where many of these requirements can begin to be satisfied.

1.3 Aims of Work

The overall aim of this work has been to explore the use of MIMD parallel computers for the dynamic simulation of chemical processes. Firstly, MIMD architectures rather than the other architecture classifications mentioned have been targeted for the following reasons:

1. Cost

MIMD architectures can be bought at relatively low entry level prices. These entry level machines contain enough processors to allow parallelism to be explored adequately and also to provide performance in excess of most modern workstations. The latest communications software also allows workstation networks to be used as a MIMD parallel resource. This is especially attractive, since parallel programs can be built and tested with standard hardware.
The utilisation of cheap MIMD machines also offers a greater incentive for industry to get involved. The average company is unlikely to spend millions of dollars on a state of the art supercomputer for more than one, if any, of its locations. However it is quite likely to support the purchase of a multiprocessor workstation, especially if it is relatively cheap and fits into their existing network with minimal disruption.

2. Largely Undeveloped Technology

Compared with other architectures, MIMD machines are still not fully utilised. In chemical engineering virtually all parallel usage of machines has been on SIMD and vector architectures. These machines have their place and are best for problems which can be solved as a large set of equations. A great deal of research into equation solving on these machines has been performed already and it seems pointless to re-explore such a well trodden area.

MIMD machines differ in that they rely much more on suitable decomposition of the problem being solved into separate subproblems. Chemical engineering has many examples which contain the possibility for such decomposition. There has been virtually no research on applying MIMD machines to chemical engineering. For this reason all of the work carried out for this thesis is targeted at MIMD architectures, although some account of the applicability of problems to other architectures is taken.

3. Heterogeneous Parallelism

Most modern computer systems are now networked and can contain a vast array of different hardware. Such a network can be thought of as a MIMD resource with various processor types. Given fast network links, simulation systems could be developed to make use of such networks with different subproblems being targeted at different but yet specific hardware. Some of the hardware on the network could be parallel resources in their own right.
Dynamic simulation has been targeted because it represents one of the most under developed and computationally expensive areas in chemical engineering. Very few commercial dynamic simulators are available, and those which are are limited in their use. Chemical plants are also highly modular, and this modularity offers possible areas for exploiting parallelism. If parallelism can be successfully used to tackle the computationally difficult aspects of simulation, it should be possible to dynamically simulate whole processes in less than real time. If this can be done robustly, the benefits for industry would be enormous.

1.3.1 The Dynamic Simulation of Distillation

The dynamic simulation of single chemical processing units can be computationally expensive in its own right. Many process units also exhibit a highly modular structure. For this reason, there exists the possibility of exploiting parallelism for them as well as complete processes. Examples of such modular processes are reactor systems, heat exchange networks and distillation.

Of these, distillation is probably the most modular. Distillation is also one of the most common pieces of equipment in use today and most computationally demanding to simulate. The main complexity is in estimating the liquid and vapour interactions taking place inside.

Given distillation's unique characteristics, it was decided to focus the initial attempts at exploiting parallelism on distillation rather than complete processes. By starting with a smaller problem, it was possible to get a feel for parallelism much quicker, and with distillation being so highly connected, a feel for the limitations on expansion of any software to use for whole processes.

The result of this initial research was a package called PDist (Parallel Distillation simulator). As it turned out, this became more of a usable tool than just a
demonstration program. This happened primarily due to the work which went into showing that although complex, parallel hardware could be utilised as simply as the sequential variety.

1.3.2 Complete Plant Simulation

After the success with PDist, it was felt that rather than simulate a different process unit in a similar manner, it would be better to tackle a larger problem. The next level up from a process unit is a process section or complete plant. A complete process contains a high degree of modularity. This modularity is of a form similar to that obtained in distillation, except with a greater number of possible connected units.

Given this specification it was decided to produce a simple prototype plant simulator. The main aim being to show that, as with distillation, a complete plant could potentially be simulated very fast given the appropriate hardware. The resulting package is called PNet and builds greatly on the work carried out with PDist. Again much of the work focuses on not only the parallelism, but also the usability of the resulting package.

1.3.3 Interaction and Usability of Parallelism

The main drawback with current packages is in the amount of time spent changing from one package to another and manipulating information manually through whatever operating system is being used. Also the more specific programs like flowsheeting packages are quite user unfriendly and are not as well developed as they could be.

The main advantage in having a comprehensive interface is that much more
complex programs can be used with little effort from the front end user. Most of the complexity can be hidden in sublayers by the application programmer with the front end user only having to provide the most essential information. Future programs are going to be much more complex due to the nature of the problems that will be tackled. Simply executing some of these programs will be complex. It is unreasonable to expect an engineer to be a packages expert. Their job is to find a way of solving a given problem. If a package is available they should be able to run it without having to read a manual and manipulate the results using the same tool they use for all the other packages they run.

With this in mind the development of PDist and PNet was extended to allow user interaction along with a standard input and graphical output format. The resulting package aims to demonstrate that it is possible to have an efficient parallel simulator which takes a standard input, can use many different models, can be interacted with dynamically, creates standard output and requires no other packages to manipulate results. A large proportion of the work performed was spent developing the interaction system and testing it on various models. The eventual success of the packages was as much down to the usability as the benefit obtained from parallelism.

1.4 Summary

The general concept of parallel computing has been introduced. This thesis aims to show how parallel computing can be used for the dynamic simulation of chemical processes. In particular distillation simulation and whole process simulation is examined. The work also aims to show that parallelism can be used simply and efficiently, even though the complexity of execution is much greater than for sequential hardware.
Chapter 2

Literature Review of Parallel Processing Research Applicable to Chemical Engineering

This chapter contains a literature review of work performed in the field of parallel processing which is either directly or indirectly applicable to chemical engineering.

2.1 Introduction

Compared with the total volume of work in the field, little research has been published on the use of parallel computers for solving chemical engineering problems. Of the research papers written, the majority are concerned with exploiting the use of vector processors such as the Cray series. Little other than speculative work has been performed on the use of SIMD and MIMD machines. The main body of the work published has also appeared during the course of this thesis. Very little work had been published prior to 1988, except for that associated with general equation solving.

The actual papers published fall into a number of categories:
• General Overviews

• General Equation Solving

• Process Simulation
  - Steady State Simulation (Flowsheeting)
  - Dynamic Simulation

• Process Synthesis

The majority of these are not directly relevant to this thesis. However, they do help to show the overall picture of where parallelism is being applied and where the actual work for this thesis fits in.

2.2 General Overviews

As with every evolving technology, there are always a number of papers dedicated to laying out the possible areas for where, and how, it can best be applied. For chemical engineering, the number of overview papers is extremely limited. Most have also been written around the middle of the last decade, when the Cray series vector processors were the most generally available source of processing power.

By far the best paper is that of McRae [2]. This is mainly concerned with modelling but provides an overview of parallel processing and its applicability for chemical engineering. Table 2.1 shows the application areas where he envisages parallel processing being applied. In the table he divides these into traditional applications and those which are emerging as a result of the availability of more computer processing power. Few, if any, papers have been published on applying parallelism to the bulk of the application areas tabled.
CHAPTER 2. LITERATURE REVIEW

Traditional Applications

| Dynamic control of large processing facilities |
| Estimation of thermodynamic properties |
| Molecular dynamics |
| Non reactive polymer processing |
| Oil field reservoir engineering |
| Process modelling and flow sheet simulation |
| Solution of FEM/FDM models of chemically reacting flows |
| Solution of very large linear programming models |

Emerging Applications

| Biomolecular dynamics |
| Computational chemistry (ab initio SCF calculations) |
| Integrated design, analysis and process optimisation |
| Non heuristic methods for batch process operation |
| Integration of process control and design |
| Robust and adaptive control |
| Model Building |
| Plant level real-time process optimisation |
| Large scale data assimilation for estimation and control |

Table 2.1: Table of Applications for Parallelism in Chemical Engineering

For these applications, McRae states that simply parallelising existing serial algorithms is the wrong course of action. He feels that a rethink of old ideas is required and that there are three main areas where improvement must come from: namely algorithms, architectures and modelling. The majority of the paper focuses on modelling and on how the various solution methods for linear and nonlinear equations can be tackled on specialised hardware. These are overviewed in Section 2.3.

The most interesting thing about the paper are McRae’s views on heterogeneous computing. He outlines plans for a flowsheet modelling environment built from a number of SUN workstations, a Warp machine (a very fast matrix multiplier), a Cray Y-MP vector processor and a specialised high speed networking system called NECTAR. The design aims to show how a flowsheets solution can be split up into a number of tasks, where each task is particularly well suited to efficient
execution on a given type of computer hardware. The idea is to reduce execution
times by distributing the tasks between the specialised hardware and use fast
communications (NECTAR) to pass the intermediate solutions between tasks.
Theoretically this should be faster than trying to solve the complete problem
on a single computer using a generic solver. The heterogeneous viewpoint is
one taken for the work in this thesis. The simulation systems described in later
chapters have been designed around using focussed applications which run on
various hardware types and work together, through communications, to provide
the overall simulation environment.

Other than the paper by McRae, which takes into account many types of parallel
architecture, the remainder of the general overview literature is aimed specifi-
cally at vector processors. The earliest of these papers is that of Stadtherr and
Vegeais[12]. They present an overview of the various performance characteris-
tics of vector processing computers and the areas in chemical engineering where
they are seen as being applicable. The most noticeable feature of the paper is
the attention given to performance differences when code written in Fortran is
compared with that written in Assembly Language. Code written in the latter
ran as much as four time faster on some machines. This highlights one of the
problems of vector processors. The majority of programmers want to use Fortran
or C. However this leaves them at the mercy of the compiler. To obtain the
maximum performance on individual vector processors, the particular hardware
must be taken into account. This usually involves programming at an Assem-
bly Language level. McRae [2] mentions similar reservations. Later papers by
Vegeais et al [13], CEP[14]and Harrison [15] reiterate the feelings of the papers
already mentioned. The only notable difference in these papers is that the use of
multiprocessor vector machines are taken more into account.

The final paper in the overview category is by Zitney et al [16]. This outlines
what the Cray series of computers are being used for in the process industries.
A number of large companies now appear to have at least one Cray machine. The major usage of these is for oil reservoir simulation and running optimised versions of existing packages such as the steady state simulators PROCESS and ASPEN PLUS [17]. Some performance results for ASPEN PLUS are given. These are described in Section 2.4.1. As with McRae, the paper also mentions the importance of heterogeneous computing. Cray want their machines to be part of the network and usable on-line rather than being detached special entities.

Perhaps the most noticeable feature of all the overview papers is that they are mainly interested in the most expensive computers of the age. Cray style machines cost many millions of dollars to buy and maintain. Nothing is said about how lower scale/cost parallelism can be used by the every day engineering company to improve its efficiency.

2.3 General Equation Solving

The solution of sets of equations is at the heart of all numerical problem solving. Over the years many numerical algorithms have been produced for solving or estimating the solution to sets of equations. Unfortunately the majority of these are geared specifically towards sequential computer architectures. This section describes some of the published work on parallel equation solving. The aim is not to provide concise overviews of actual algorithms, but to outline the general methods which are being tailored for parallel execution and to provide pointers for further reading. Where the theory is particularly relevant to dynamic simulation and the work described in later chapters, more explanation is given. This is particularly the case where linear and non-linear systems are concerned.
2.3.1 Linear Algebraic Equations

The solution of most forms of equations results in a need to solve a linear set of algebraic equations. For this reason the efficiency of the linear solution step can be critical to an overall solution methods performance.

Sets of linear equations can be represented by the matrix formula:

\[ A \cdot x = b \quad (2.1) \]

where \( A \) is the square matrix of equation coefficients, \( x \) is the vector of solutions and \( b \) is the vector of equation right hand sides. The numerical solution of these equations is usually carried out in one of two ways: either using an elimination method or an iterative method.

Elimination Methods

Elimination methods are probably the most commonly used. The rows and columns of 2.1 are manipulated to eliminate particular elements of the matrices. The aim is to produce a more structured and manageable form for \( A \).

The best known of these methods is Gaussian Elimination. The method manipulates the matrix \( A \) into an upper triangular form \( U \). Starting at the bottom of \( U \), the solution vector \( x \) can be evaluated simply by back substitution. Sometimes the elimination step is complicated by the presence of zero elements on the main diagonal or numbers which cause round-off error problems if used for elimination. This usually results in the need for rows (partial pivoting) or rows and columns (full pivoting) to be reordered so as to have a particularly desirable element in the diagonal position.
The two other popular elimination methods are Gauss-Jordan Elimination and L.U. (Cholesky) Decomposition. Gauss-Jordan elimination is similar to Gaussian Elimination except that the matrix manipulation is designed to produce the inverse matrix of $A$ as well as the solution. With Gauss-Jordan elimination the solution is obtained directly. No back substitution step is required. Back substitution is a highly sequential operation and not trivial to parallelise. Gauss-Jordan is thus potentially more parallelisable. Unfortunately, Gauss-Jordan has a much higher operation count than Gaussian Elimination.

L.U Decomposition involves decomposing 2.1 to the form:

$$A \cdot x = L \cdot U \cdot x = b \quad (2.2)$$

where $L$ is a lower triangular matrix and $U$ is an upper triangular matrix. Once decomposed, the overall solution is obtained by solving $L \cdot y = b$ using forward substitution followed by solving $U \cdot x = y$, using back substitution as with the Gaussian Elimination method.

**Iterative Methods**

Iterative, or relaxation, methods take equation 2.1 and rearrange the structure such that from an initial estimate of $x$, a series of evaluations can be performed giving rise to a new estimate of $x$. This can then be repeated until the value of $x$ converges to a predefined tolerance. The main difference between methods is in the evaluation path used and the way in which the newly updated estimate $x$ are used.

The two most common relaxation methods are the Gauss-Jacobi and Gauss-Seidel methods. In each of these equation 2.1 is decomposed as follows:
where \( L \) is the lower triangular matrix of \( A \) with zeros on the diagonal, \( D \) is the diagonal part of \( A \) and \( U \) is the upper triangle of \( A \) with zeros on the diagonal.

For the Gauss-Jacobi method 2.3 is rearranged to the form:

\[
D \cdot x^{(k)} = -(L + U) \cdot x^{(k-1)} + b
\]  

(2.4)

where the superscript on \( x \) refers to the iteration number on which the value was generated. The feature of the Gauss-Jacobi method is that at each iteration only values from the previous iteration are used.

In contrast to this, the Gauss-Seidel method uses the following rearrangement of 2.3:

\[
(L + D) \cdot x^{(k)} = -U \cdot x^{(k-1)} + b
\]  

(2.5)

In this case the presence of \( L \) on the left hand side causes the new values \( x^{(k)} \) to be incorporated in the iteration as soon as they are generated. The Gauss-Seidel method is usually faster to converge than the Gauss-Jacobi method, but this is not guaranteed.

Further adaptations of the Gauss-Seidel method are made by adding a relaxation parameter \( \omega \). This acts as an acceleration parameter to the method. When \( \omega \) is in the range \([0, 1]\) the method is said to be under-relaxed. When \( \omega \) is in the range \([1, 2]\) the method is said to be over-relaxed and gives rise to the SOR (Simultaneous Over-Relaxation) method. There are a wide variety of methods
for choosing the value of \( \omega \) and manipulating it throughout a solution.

Parallel Solution

The bulk of the literature examines the parallelisation of the above methods. Of these, most are highly theoretical. The algorithms presented ignore the overheads associated with memory access, input/output, data management and interprocessor communication. The algorithms are thus targeted for many more processors than can realistically be used in practice. The number of papers which actually present implementation results is minimal.

All the algorithms presented are highly dependent on the particular hardware. For vector processors the main aim is to reduce the number of numerical operations that must be performed sequentially and to maximise the number and size of vector operations. For MIMD style machines, the aim is to maximise the amount of parallel computation which can be performed. For machines such as vector multiprocessors, the aim is to find a trade off between the two. Miranker [18], Poole and Voigt [19], and Heller [20], provide overviews of the work on linear systems up until the end of the 70s. For elimination methods the Gauss-Jordan method appears to be the most suited to MIMD style architectures. This is mainly down to the highly concurrent nature of the method. The only difficulty is when pivoting is required. However, the operation count for implementing this is stated as being relatively insignificant when comparing the algorithm to that for other elimination methods. For vector processors, Gaussian elimination appears better. Gaussian elimination requires much fewer operations than Gauss-Jordan on a sequential computer and contains similar operations from a vectorisation viewpoint.

For iterative methods there are much fewer algorithms. The Gauss-Jacobi method parallelises simply on MIMD machines. However, the convergence of the method
is slow and solution via another method is probably desirable. A variant of the SOR method is outlined. Instead of the calculation sequence being highly sequential, the evaluations and solution updates are performed chaotically. The overall approach is very Gauss-Jacobi like. The main difference is that solution estimates are used as soon as they are re-evaluated. Problems associated with algorithm control are mentioned, but the overhead associated with implementing the control mechanism is not.

The most noticeable omission from this early work are algorithms for solving sparse systems. Sparseness causes real problems for vector processors. Vector operations become inefficient as vector lengths reduce. For any conventional solution method it is difficult to locate vectors of any length which can be usefully operated on. From a MIMD viewpoint, relaxation methods are viewed as the most promising. The very nature of these methods means that they effectively deal with sparseness.

Since 1980, a number of other papers have been published which further expand on this work. The algorithms described are extremely technical and their explanation serves no real purpose for this thesis. For those interested the various papers found are listed. For iterative methods there are the papers by Dekker [21], Reed and Patrick [22], Baker et al [23] and Vorst [24]. For elimination methods the number of relevant papers is much larger. These cover a much wider range of equation structures and methods. Gaussian elimination is however particularly popular. The papers worth examining are those by Meier [25], Dongarra and Johnsson [26], Liu [34], Chu and George [28], O'Leary [29], Vorst [30], Bjorstad [31], Cosnard et al [32] and Marrakchi and Robert [33].

A number of the papers listed make extensive use of the BLAS (Basic Linear Algebra Subprograms), either directly or indirectly through the LINPACK package. The BLAS routines provide basic operations such as vector update, dot product, vector scaling, rank 1 updates, triangular solver etc. These have been optimised
for use on various specialised hardware. In particular vector processors. Descrip-
tions of BLAS are given by Lawson et al[7], Dongarra et al[8], Croz and Mayes[9],
Mayes [11]. and Dongarra et al[10]. Recent developments are towards providing
similar routines for use with MIMD machines. Phillips [35] presents some work
towards achieving this aim.

Of more interest to this thesis is the work published relating to the solution of
sparse systems. The linear equations produced during the solution of steady
state and dynamic process simulation problems tend to be of this type. On
sequential machines, specialised algorithms are usually used to indirectly index
matrix entries and thus reduce the overall storage requirement for the matrices in
question. The sparse methods then work with this reduced data structure. For
vector and parallel machines this causes a number of problems. Vector processors
do not cope particularly well with vectors which are not continuous in memory.
Some modern processors have special hardware for this, but the operation is still
less efficient than a conventional vector operation. By leaving the matrices as full,
larger and more continuous vectors can be operated on. However, a large amount
of the vector operation is wasted on zero entries. For MIMD machines, these
problems are enhanced. Parallelising the specialised sparse solvers is difficult
due to the highly sequential nature of the mechanisms used to cope with sparse
storage. If this is removed, the same problems associated with the solution of
dense systems are experienced. Additionally, with sparse systems it is more
difficult to provide each processor with an even amount of work. Particularly
when the equations are of an uneven structure.

Heath et al [36] present and review the recent developments in parallel algorithms
for sparse systems. As well as examining the numerical factorisation of the equa-
tions, algorithms for the other stages in solution are also reviewed. The overall
conclusion of the paper is not particularly hopeful. Of the papers reviewed, none
appear to have produced any results which stand out as exciting. The most
successful results having been produced on shared memory MIMD machines.

Of the algorithms overviewed by Heath, the class known as Frontal methods appear to have been those targeted for use with chemical engineering problems. In particular they have been applied to the solution of steady state flowsheeting problems on Crays, see Section 2.4.1. Frontal methods were developed as a means of saving storage space and working around the problems associated with the indirect addressing of vectors. The method works by keeping only a small amount of the equation matrix in main memory at any given time. This matrix is called the Frontal matrix. It is stored in full form, and thus vector operations on the matrix do not suffer from indirect addressing. The solution method begins at the top of the equation matrix and works downwards. At each row, the frontal matrix is expanded to take into account the variable additions from the current row. Once all of the rows which contain a particular variable have been added, an elimination on the Frontal matrix is performed to eliminate that variable from the frontal matrix. The deleted pivot row is stored in memory for use with back substitution at the end to obtain the overall solution. This continues until the complete matrix has been processed. The method is most effective when the Frontal matrix is kept small. To facilitate this, the overall equation matrix is reordered to a more amenable form. If the Frontal matrix does increase in size, its solution on multiple processors becomes feasible. When applied to multiprocessor machines the Frontal method becomes known as the Multi-Frontal method. More information on Frontal methods can be found in Heath [36] and Dave and Duff [37].

As yet no one method appears to be radically more efficient than any other. The method implementations are highly technical and very hardware specific, with the main efficiencies arising from clever encoding of the algorithms. Their realistic use for major applications is only going to be through standard library packages such as the BLAS and LINPACK.
2.3.2 Other Equation Types

This section moves on to consider the solution of other types of equations. These are all dealt with together. Linear systems have been treated separately, since the main body of solution methods for the equations now described, usually require the solution of a linear system of equations at some point. Of main concern to this thesis is the solution of nonlinear, ordinary differential and differential algebraic equations. These are the most directly relevant to dynamic simulation. Of less interest here is the solution of partial differential equations. These are however touched on because of their importance for Oil Reservoir simulation.

The numerical solution of almost all forms of differential equation systems involves replacing the differential terms with discrete approximations. This produces a discretised set of equations. The method used to discretise the equations varies from method to method. However, the resulting system of equations is either a linear or nonlinear system. The solution of linear systems has already been overviewed. The main body of this section is thus dedicated to the solution of nonlinear equations.

All nonlinear equation solution methods are essentially iterative. Some of these iterative methods, the fixed point methods, are very similar to the Gauss-Jacobi and Gauss-Seidel methods for linear systems. The parallelisation of these methods is thus analogous to that of their linear counterparts. The use of these has been particularly targeted at partial differential equations, where blocks of mesh calculations can be solved in a Gauss-Jacobi fashion. Hart and McCormick [38, 39] and Saltz and Naik [40] present parallelisation work in this area. By far the most popular of the other nonlinear solution methods are the Newton and Quasi-Newton methods. These methods work by taking a linear approximation of the functions at an estimate of the root. This approximation is then used to produce a new estimate which is hopefully more accurate. This linear approximation is
taken as the tangent to each equation at the root estimate. The linear approximation thus contains an estimate of the first derivative of each function. The general update formula for Newton's method is shown below. In the equation $z$ is the vector of dependent variables, $J$ is the Jacobian matrix and $f$ is the vector of function values evaluated at $z$. The superscript describes the iteration to which the values refer.

$$z^{(k)} = z^{(k-1)} - J(z^{(k-1)})^{-1} \cdot f(z^{(k-1)})$$  \hspace{1cm} (2.6)

The main feature of the method is that to obtain the new estimate of the root, the linear system $J \cdot (z^{(k)} - z^{(k-1)}) = -f$ must be solved at each iteration. This linear solution phase presents a problem for parallelism as previously discussed. However, as well as the linear solution phase, the calculation of the Jacobian $J$ and the function values $f$ are also required. These can be calculated easily in parallel and it is this which the majority of published work has focussed on. Quasi-Newton methods work in a similar manner, except that the Jacobian is regularly estimated from historic iteration data rather than being rigorously calculated. The advantage is reduced calculation load. The disadvantages are reduced convergence speed and reduced parallelisability.

In many chemical engineering problems, the calculation of the Jacobian and function value matrices can be the most computationally expensive part of the overall solution. This is particularly the case for process simulation, where some of the function evaluations require complex physical property calculations. Of the published work, most describe an algorithm for partitioning the function and Jacobian evaluations between processors. The papers by McRae [2], Juarez and Pantelides [41,42] and Ponton et al [43] present a good overview of the problem. In the last two of these, actual implementation results are published. In both the overall time spent in performing function and Jacobian evaluations was sig-
significantly reduced. However, the examples used were not particularly complex and the linear solution step was found to dominate the overall solution. Ponton et al [43] attempted to avoid this problem by parallelising the linear solver. However, communication overheads were found to be significant and little overall benefit was obtained. The best results were obtained when a full sparse solver was used for the linear solution phase. Although these results do not look particularly promising, for larger and more complex systems the efficiency is likely to be greatly improved.

Overall, the efficient solution of nonlinear and differential systems on parallel computers is extremely difficult. This is primarily due to the requirement for the solution of a linear sub system and the difficulties associated with distributing the calculation load evenly between processors. In this area there is still a lot of work to be done. For many problems this difficulty can be avoided by geometrically splitting the overall problem into a number of subproblems, each of which can be solved on separate processors using efficient and well tested sequential algorithms. The majority of the published work on parallelism and chemical engineering uses this partitioning approach.

2.4 Process Simulation

The simulation of chemical processes comes in two forms: steady state (flowsheeting) and dynamic. In either case, the most common solution methods are categorised as being either “equations based” or “modular based”. Equation based methods take the set of equations which model the process and solve them as a single set. Modular based methods split the overall process into blocks based on process topology and solve each block separately. On sequential machines, the blocks are usually solved in the order they appear in the real process, with the newly calculated output from one block being used as the input to the next.
When recycles are present this calculation sequence is repeated until the overall process has converged. In this form the method is known as the “sequential modular” method. This section reviews the work which has been published on the parallelisation of process simulation via these methods. For the moment the details of equation based and modular based methods are not greatly elaborated on. A detailed description of the methodologies and their applicability for parallelism is deferred until Chapter 3.

2.4.1 Steady State Simulation (Flowsheeting)

Flowsheeting requires the solution of sets of nonlinear equations. The structural nature of chemical processes results in these equations being predominantly block diagonal with a number of off-diagonal elements. The off-diagonal elements represent the variables shared by the equations of different process units which are connected. The equations are also sparse.

For equation based methods, the only option is to use a parallel nonlinear solver. The various ways of achieving this have already been discussed. In the literature, the main focus of attention has been towards the parallelisation of the linear solution phase. For the most part the overhead associated with the Jacobian and function evaluations has been ignored. In contrast with the work on modular methods, where for the most part the literature is concerned mainly with distributing the function and Jacobian evaluations.

For the linear solution phase, the most significant work has been performed by Zitney and Stadtherr [44,45]. All of the work methods to enhance solution times on the Cray series. In section 2.3.1, Frontal methods were designed to solve on overall memory usage. The papers examine the performance of parallel solvers against the general
sparsity solver LU1SOL used for an experimental equation based simulator called SEQUEL-II. For the actual nonlinear equations, the thermodynamic properties were estimated using Peng-Robinson and a Quasi-Newton method was used to solve the system. Peng-Robinson is not particularly complex to compute. For the examples run, this meant that the linear solution phase contributed a significant part to the overall computational load. The results show that the highly optimised frontal solvers at best outperformed LU1SOL by a factor of 10. The overall result is encouraging. However, little is mentioned about the amount of optimisation which has gone into the LU1SOL solver. The frontal solvers developed have parts written in assembly language. This alone can enhance performance greatly on vector processors. The modelling approach used is also designed to minimise somewhat the overhead associated with physical property and Jacobian/function evaluations. Complex physical property evaluation is usually highly iterative and vector lengths are greatly dependent on the number of components. The vectorisation of these is thus difficult when component numbers are small. The really time consuming flowsheets to solve are usually those which are highly non-ideal. In such cases the physical property calculations are likely to overwhelm the linear solution phase quite significantly.

Other work related to this general area of optimising the linear solution phase for flowsheets are presented by Coon and Stadtherr [46], Wait and Landauro [47] and O'Neill et al[48]. The latter two of these focus particularly on distillation and the banded linear systems produced by the Napthali and Sandholm [49] approach to simulation. No implementation results are shown for these.

Of the equation based method papers, the only one to really target the overall solution from both a linear and Jacobian/function evaluation viewpoint is McRae [2]. He outlines plans for what is essentially a heterogeneous parallel nonlinear equation solver. The various parts of the solution are split between different hardware platforms: the linear solution phase is carried out using a Systolic Array
(Warp) machine with the Jacobian, function and physical property evaluations being performed on a parallel processor. An extremely fast communications system is then used to pass the various solution parameters between machines. In the paper, the machine is only hypothetical. However, since then McRae [50] has presented results from the use of such a machine for the atmospheric modelling of pollution effects around Los Angeles. The overall approach is an extremely interesting one. However, very few companies could warrant the purchase the connection mechanism, never mind the complete machine.

The remainder of the literature on flowsheeting is concerned with modular methods. Again Zitney [51,52] has performed a large proportion of the work related to their adaptation for vector multiprocessors. The earliest work was not really targeted at optimising the overall simulator, but at parts of it, in particular the solution of the complex distillation modules. Again frontal methods are used. This time the Harwell MA28 sparse linear solver used in ASPEN PLUS is replaced. As before, overall speedups of around 10 were achieved. For the actual linear solution phase, speedups as great as 80 were obtained for some problems. This suggests that the function and Jacobian evaluations were dominant during the overall solution phase. Harrison [53] also presents implementation results of a simulator FLOWTRAN [54] on similar hardware. No actual optimisation is carried out in this case however.

The other main contribution to the literature on sequential modular flowsheeting is by Best [55,56]. In this case the hardware targeted is of the MIMD variety, in particular a transputer based, distributed memory machine. The papers examine the adaptation of a two-tier solution method for flowsheeting developed by Johns and Vadhwana [57]. This two-tier method uses parametric approximations of the complex physical properties to speed up the overall rate at which the flowsheet can be solved. Every so many iterations, the parameters used for the approximations are rigorously updated. The approach taken by Best is to distribute the
overall process units among the processors available and have them calculate in a sequential modular fashion. This is a highly sequential operation and for the majority of the time only one processor is actually actively solving a particular unit. The rest are idle. Best uses this idle time to update the parameters used by the simple models. A series of actual timings results are presented. From these it is difficult to draw any real conclusions, especially since 10% greater speedup than that theoretically possible was obtained for some of the simulation runs. This was put down to hardware fluctuations between runs, but given that only four processors were used it would not have been hard to produce a series of averages. Again results look promising, but a much more rigorous analysis of the method is required.

The remaining publication looks at a hybrid of the equation based and sequential modular methods. The "simultaneous modular" approach takes the flowsheet and splits it into sections based on process topology. Instead of solving these in a sequential manner, a set of nonlinear equations is produced which describe the connections of these blocks. These connection equations are then solved using a standard nonlinear solver. The partitioned process blocks are used to evaluate the connection functions and to produce the connection equations Jacobian. This Jacobian is usually calculated by perturbation of the inputs to the process block and analysis of the resulting changes in the outputs. The overall methodology is again close to that of the distributed function and Jacobian solution method for nonlinear systems. Unlike with straightforward nonlinear solution, the equation system being solved is much smaller. The linear solution phase is thus much less of a performance bottleneck. Chimowitz and Bielinis [58] explore the use of this approach. Unfortunately no actual implementation results are presented.
2.4.2 Dynamic Simulation

The dynamic simulation of chemical processes requires the solution of sets of differential and algebraic equations, DAEs. For equation based methods, the DAEs can be solved in two ways: either using a single DAE solver or by solving the differential and algebraic parts separately. Either way, the solution of a set of nonlinear equations is eventually required. All of the parallel nonlinear solution methods described previously are applicable here. Modular methods tackle the overall problem in a different way. The main problem found when trying to modularise the solution of flowsheets was in partitioning the overall equations into blocks which were not sequentially dependent on each other. The main reason for this is that at the start of a flowsheet calculation a good estimate of the overall solution is not known. With dynamic simulation this is not the case. Dynamic simulations progress in discrete time steps. Over a given time step the actual changes in the connection variables between process blocks is relatively small. The starting point for a given time step thus provides an excellent estimate of the solution. Modular methods take advantage of this feature by separating the process into decoupled blocks. Each block can then be solved to estimate the output for the next time step either explicitly using the current input or implicitly by using an estimate of the input at the next time step calculated from historical data. In the later case, all the blocks usually repeat this step until a converged state is reached. This iterative process is normally performed in a sequential modular manner to reduce the amount of connection estimation required. It is however possible to completely parallelise this approach. A detailed analysis of modular methods and their parallelisability is given in the next chapter.

Little work has been published on parallel dynamic process simulation. A recent survey by Moe and Hertzberg [59] confirms this. Of that published, most is concerned with the simulation of distillation. No literature was found on parallel implementations of complete process simulators. The most significant work on
equation based methods has been carried out by Skjellum [60], Skjellum et al [61] and Secchi et al [62,63]. In the earliest work by Skjellum [60], a parallelised version of the DAE solver DASSL, [64] was used to simulate 7 connected distillation columns. For the Jacobian and function evaluation stage the speedup obtained was around 100 for 128 processors. However, the linear solution step was found to be the main bottleneck and the overall speedup obtained was around 5 when compared to an efficient sequential algorithm. The later work by Skjellum et al[61] and Secchi et al [62,63] considers the use of Waveform Relaxation to solve similar problems. Waveform Relaxation is analogous to the relaxation methods used for linear systems. In this case subsystems of equations are iterated upon rather than individual variables. The results from this work has been more encouraging that that obtained with parallel DASSL. For the larger column examples run, speedups as high as 60 were obtained over the best sequential algorithm. The problems used here were relatively simple from a thermodynamic viewpoint. It is likely that greater efficiency would be obtained if more complex models were used. This may however not prove to be the case. The work in this thesis has shown that for more complex models, the even loading of processors becomes more difficult to achieve, especially when large dynamic changes are only occurring in certain parts of the process. Other related work in this area has been performed by Lin and McGreavy[65]. Again a distributed Jacobian/function evaluation approach has been used. This time however, actual implementation has only been simulated using the multitasking features of the ADA language.

For modular methods there is a similar lack of literature. The earliest work is that of Ponton et al [66]. A number of Acorn BBC computers were connected via a LAN (Local Area Network) to produce what could be regarded as a distributed memory MIMD machine. This was used to simulate a series of simple processes including an LPG cargo plant and a four bed pressure swing absorption unit. The overall result was not not that the simulations were particularly fast, but that the approach was possible. For the hardware used, communication times were highly
significant and no great speedup could have been expected. Of the literature reviewed, this is the only one to contain work on actual parallel modular process simulation, where different unit types are involve.

The remainder of the modular literature is again concerned with distillation. Cera [67] describes the parallelisation of the distillation algorithm from a dynamic simulator called DYFLO. The models used are simple and the integration method an explicit one. The best result obtained was a speedup of around 8 using 14 processors. The machine used was a BBN Butterfly which is a shared memory machine. Watanabe [68] presents similar work but at a much smaller scale. The rest of the published work on parallel modular process simulation methods has been by this author and associated co-workers [69], [70], [43,71], [72]. The work goes much farther than that published by other authors, both from a complexity and usability viewpoint. This thesis is designed to fill a large gap in the literature concerning parallel modular process simulation. The resulting modular simulator, PNet, is thought to be the first of its kind. No literature on process simulators of this type was found by this author or by Moe and Hertzberg [59] who have published a recent survey of the field.

2.5 Process Synthesis

Process synthesis is the automatic generation of one or more chemical processes based on an initial design specification. The task is an exceptionally complex one. For any given design problem, there are usually many different ways of solving it. Some of these will be attractive from one perspective but not from another: e.g. cheap but dangerous. The overall problem is thus one of finding alternatives and examining the viability of each. For each of these alternatives, some may be invalid for obvious reasons and discarded at an early stage. For those alternatives left, a process design study must be carried out. This design study itself may
result in whole series of other alternatives being examined. This method of locating and evaluating alternatives is extremely computationally demanding. Given that the rules of search and evaluation are optimised, there is no obvious way in which the computational requirement can be reduced. The only real solution to enhancing performance is thus to use parallelism. The nature of the synthesis methodology is highly modular. Many design evaluations can conceptually be carried out at the same time.

The main publications in this area are by Fraga et al [73] and Fraga and McKinnon [74], [75], [76], [77]. A parallel (MIMD) synthesis package called CHiPs has been produced for generating costed distillation column sequences given an initial input stream specification. The package designs columns via a comprehensive unit model interface. This allows many different models to be used, ranging from simple ideal models to highly complex azeotropic models. The other main feature of CHiPs in its ability to automatically consider heat integration during synthesis. The parallel methodology used is a master/slave approach, where design calculations are distributed amongst the available slave processors. To help avoid bottleneck problems with the master, some of the search algorithm is parallelised as well. CHiPs has been run on a number of architecture types ranging from transputer and i860 based Meiko Computing Surfaces to networks of SUN workstations. Significant speedups over efficient sequential algorithms have been obtained on all of these platforms using simple thermodynamic property estimation. Work is now underway on using complex thermodynamic estimation to allow the synthesis of separation systems which contain azeotropic components. For these more complex synthesis problems the amount of design required is enhanced and results should continue to improve. The only real problem with using extra processors is in finding enough for them to do. All problems have an upper limit of processors which can usefully be utilised.

The only other parallel work on synthesis found has been by Zitney et al[78].
Again Cray computers are the main target for this work. The paper describes the authors views on the provision of a computer-integrated environment for process operations. This takes into account steady state and dynamic simulation, expert systems etc. There are no actual implementation results. The paper is mentioned here purely for interest.

2.6 Summary

The literature on parallel processing for chemical engineering is extremely limited. Of that published, most is hypothetical and has never been implemented on real hardware. It is also noticeable, that for all of the topic areas except equation solving, the literature that exists is being published by a small and select group. The overall field is thus very open. This thesis aims to fill a large gap in the dynamic simulation area for MIMD machines.
Chapter 3

Dynamic Process Simulation: Theory, Parallelism and Application to Dynamic Distillation Simulation

The dynamic simulation of chemical processes is computationally expensive. It is therefore not surprising that very few dynamic simulators have models which contain adequate complexity to be truly useful. Coupled with this, the nature of the equation systems which model the processes make their solution non-trivial. Until recently, vector supercomputers offered the main alternative to using standard sequential hardware to solve models. However, the advent of low cost/high performance MIMD machines has opened a new source for obtaining the raw processing power required. The problem remaining is how can MIMD machines be exploited for the types of problem we wish to solve.

This chapter introduces the theory behind dynamic process simulation and examines the possibility of using MIMD parallel computers to obtain improved performance. Descriptions of the implementations of this theory are given in Chapters 4 and 5. Of the methods described, a number are highlighted as suitable for parallel execution. The chapter ends with a description of the initial
work carried out on the parallel simulation of distillation columns using one of these methods and of the models developed.

3.1 Dynamic Simulation: Theory and Parallelism

The dynamic simulation of a chemical process requires the solution of a number of differential and algebraic equations (DAEs). These DAEs have a number of characteristics:

- "stiffness"

A stiff system of equations occurs when there are two or more very different time scales of the independent variable on which the dependent variables are changing. When explicit integration techniques are applied to approximate the solution, stability concerns force the integration time step to become small even when a very much larger one could be used from an accuracy point of view. The term stiff systems comes from their application in analysing the motion of spring and mass systems having large spring constants [79].

The solution to stiffness is to use implicit integration methods. Unfortunately these tend to be iterative and more computationally demanding.

- Highly Nonlinear

This is especially true when complex physical property, kinetic and hydraulic relationships are used.

- Sparse

The equations representing each unit share very few variables with those of other units. The equations representing single units can also be sparse, as
in a distillation column. The resulting system of equations is usually over
95 % sparse and requires the use of efficient algorithms which take into
account this sparsity. Sparsity is also a major problem when it comes to
parallelisation, see Chapter 2.

- Discontinuities

During dynamic simulation, a situation often arises where the original equa-
tions describing the system are no longer valid. Locating and handling these
can be extremely difficult.

A set of DAEs can be defined by:

\[ f(\dot{z}, z, v, t) = 0 \]  \hspace{1cm} (3.1)

where \( f \) is the vector of all equations, \( z \) and \( \dot{z} \) are the vector of variables and time
derivatives respectively, \( v \) is the vector of external inputs and \( t \) is time.

The equations making up 3.1 can also be represented by a set of equation vectors,
\( f_i \), which represent the block of equations strictly applying to a particular unit \( i \)
in the overall chemical process. For a unit \( i \) of \( m \), there is a vector of associated
functions \( f_i \), where

\[ f_i(\dot{z}_i, z_i, u_i, v_i, t) = 0 \]  \hspace{1cm} (3.2)

such that 3.1 becomes

\[ f(\dot{z}, z, v, t) \equiv \{ f_1, f_2, \ldots, f_m \} = 0 \]  \hspace{1cm} (3.3)
$z_i$ are the state variables of unit $i$, $v_i$ are the subset of inputs $v$ which apply to unit $i$ and $u_i$ are the input variables from other blocks $f_j : j = [1, m], j \neq i$. $u_i$ is essentially a subset of $z$.

There are a number of different approaches used to solve such systems. These are usually categorised as either being "equation" or "modular" based. Overviews of the various approaches are given by Hillestad and Hertzberg[80]. The terminology comes from methods used initially for solving the large nonlinear systems associated with flowsheeting. Shacham et al [81] provide an overview of the methods and terminology used by flowsheeting systems.

### 3.1.1 Equation-Based Methods

Equation-based solution methods treat the equations as a whole and solve them using a single solver. The most common method of solution involves using a difference approximation to the differential terms and solving the resulting set of nonlinear equations over a single global time step.

The main disadvantage of using the equation based approach is the requirement to discretise all the equations at the same point. This results in the stepsize for all equations being dependent on the fastest moving variables in the system. Stiffness is avoided by using implicit integration algorithms or so called "multi rate" methods. For multi rate methods the equations are split into slow and fast moving subsystems which can be solved separately using different integration methods.

From a parallel processing viewpoint, the main problem is in solving the set of nonlinear equations produced by whichever integration algorithm is employed. After discretisation we are left with the following:
\[ f(z, z^0, v) = 0 \]  

(3.4)

where \( z^0 \) are the known state variables at time \( t \).

These nonlinear equations can be solved in a variety of ways. The most common methods are Newton or Quasi-Newton methods. Newton’s method for nonlinear systems uses the update formula shown below.

\[
z^{(k)} = z^{(k-1)} - J(z^{(k-1)})^{-1} \cdot f(z^{(k-1)})
\]  

(3.5)

Here \( z \) is the vector of dependent variables, \( J \) is the Jacobian matrix and \( f \) is the vector of function values evaluated at \( z \). The superscript describes the iteration to which the values refer.

The general algorithm for a system of \( n \) equations involves evaluating the \( n \) function values \( f \) and the Jacobian \( J \) which usually requires \( n^2 \) function evaluations. Once evaluated the linear system shown below is solved for \( (z^{(k)} - z^{(k-1)}) \) and \( z^{(k)} \) is updated. This is repeated until \( (z^{(k)} - z^{(k-1)}) \) is less than a specified tolerance.

\[
J \cdot (z^{(k)} - z^{(k-1)}) = -f(z^{(k-1)})
\]  

(3.6)

Overall the function and Jacobian values require \( n^2 + n \) function evaluations per iteration. Quasi-Newton methods avoid this overhead by estimating some of the Jacobian calculations rather than fully evaluating them. Usually these methods can reduce the number of function evaluations to \( n \).

Newton’s method works well, especially when a good estimate of the solution is available. In such circumstances the convergence is quadratic. Quasi-Newton
methods also display good convergence, but less so than straight forward Newton. The convergence of Quasi-Newton methods is usually referred to as super linear. In both methods the effort required for the linear solution phase per iteration is the same. It is usually advantageous to accept slower convergence, given the reduction in overall function evaluations required.

In terms of parallelising such methods, two realistic options are available. These are:

1. Distribute $f$ and $J$ calculations and solve the linear system locally.
2. Distribute $f$ and $J$ calculations and parallelise the linear solver.

Fraga et al[43] have tried these different approaches for Newton's method. Method 1 where the function and Jacobian calculation alone is partitioned produces a speedup with added processors. This speedup however, decays more rapidly than expected. The reason is that the function and Jacobian processors are idle during the linear solution phase. This is an example of Amdahls Law [82], where a sequential portion of the solution limits the possible speedup. In method 2, where the linear solution phase is parallelised, communication delays due to implementing the solver were also found to limit speedup. In practice the best results were produced using method 1 with a full sparse linear solver.

The parallel Newton approach obviously suffers from the requirement for the repeated solution of the linear system. It is going to be most successful when $f$ and $J$ evaluations are expensive and dominate the overall calculations. This is likely to be the case in chemical processing where physical property evaluation is usually the dominant calculation. The amount of calculation can be reduced using Quasi-Newton style methods, but this only reduces the amount of distributed evaluations and does not affect the linear solution phase.
The other main difficulty is in distributing the calculations. The equations being solved are very patchy and require differing amounts of computation. Assigning blocks to processors so as to assure load balancing and stability is not simple.

Other approaches to solving sets of non-linear equations are "fixed-point" methods. These involve rearranging the equations being solved to express each one in terms of its respective variables. Then, starting at one end and working towards the other, each variable is calculated sequentially. Depending on whether old or newly evaluated variables are used, the methods are analogous to the Gauss-Jacobi and Gauss-Seidel methods for linear systems respectively.

Parallelising such methods is simple in the Gauss-Jacobi case. Unfortunately the sequential nature of the Gauss-Seidel approach offers less possibilities. The main advantage of fixed point methods is that they effectively deal with sparseness. The main difficulty is in rearranging the equations into the form required. This is especially difficult when subsystems of equations are highly dependent on one another as in vapour/liquid equilibrium systems. Convergence is also slower than Newton’s method and not guaranteed. At first glance fixed point methods do not look very promising, however as shall be explained in the next section, a similar approach can be used at a much coarser level.

Overall equation-based methods should be parallelisable. However, the aims of the project are to provide robustness as well as speed. Using a single method to solve a complete process is always going to be problematic, since a general method is being applied to a collection of equation blocks which each have different characteristics defined by the unit they represent. It is also difficult to cope with discontinuities, especially in a parallel environment.

Finally, a full solver does not easily allow the inclusion of existing simulation code. A lot of modelling code already exists, in particular reactor, distillation, flash and kinetic models. Most will have been solved using an integration method
specifically designed for the local problem. It would be a undesirable to throw away such past work. The solution to using this code lies in modular integration methods. The next section explains what they are and how they provide the functionality and robustness we are looking for.

3.1.2 Modular Based Methods

Modular based methods take the set of equations represented by equation 3.1 and divide them up into sub blocks based on process topology. For each of these sub blocks the equations can be solved to provide a set of outputs given a set of inputs.

To solve the complete system there are two possible modular approaches. These are:

1. simultaneous modular

   The sub blocks connections are all integrated by a single integration routine. The sub block models are used to provide derivatives for the solver. This is usually done by perturbation of the input variables.

2. sequential/uncoupled/independent modular

   Sub blocks are treated as individual simulators with their own integration routines. Input variables are controlled in a variety of ways depending on the connection strategy employed. Similarly the name depends on the connection strategy used. Each sub block is integrated over a common time horizon. Each block is able to use whichever method and integration time step it requires within this time period. Intermediate input values are usually provided by interpolation polynomials.

Method 1 is similar to the parallel Newton's method where the function and Ja-
cobian evaluations are distributed. This time the system being solved is smaller, since only the connection equations are being solved. However a parallel method would still suffer similar problems. This method is essentially a half way house between equation-based and full modular methods.

Method 2 is much more promising. The key is to be able to suitably decompose the problem to allow all of the blocks to be independent of each other over a given time horizon.

Using the vector notation from before, at each unit i, new values for the output and state variables are being evaluated from the current state variables and the newly updated input variables. Hence

\[ z_i = f(z_i, z_i^2, u_i, v_i) \] (3.7)

For a process with no recycles as shown in figure 3.1, \( u_i \) is not affected by any future evaluations of \( z_i \). In this case the process may be solved in a single pass by starting at block A and working sequentially through blocks B, C and D. Usually there are quite a few recycles in a given process and this no longer holds. Figure 3.2 shows such a process. Here there is no sequential path of calculation which avoids using an input stream which is unaffected by this and future block evaluations. In such cases, the unknown inputs have to be estimated and multiple iterations performed. After each pass the recycle streams are updated and this continues until convergence is obtained. The method now involves iterating on a number of unknown streams. These are called "tear" streams and are key to the modular solution methodology. Usually the updated tear stream values are calculated using an extrapolation algorithm. The method described here is the classical sequential modular approach.

Although this approach suits serialism, it does not map well to parallelism. By
tearing only a few streams, there is too much inter unit dependency, and hence blocks cannot usefully run concurrently.

The solution is to tear all of the streams connecting the sub blocks together. In order to allow each block to operate concurrently the tear stream values over the time horizon\((TH)\), \([t, t + TH]\) must be estimated. This can be performed using a suitable extrapolation method. All of the sub blocks can now be run in parallel. Once complete the estimated tear variables can be compared with those calculated. Depending on the error another estimate of the tear variables can be performed followed by another iteration, or the time horizon can be reduced for the next integration.

This approach is far from new. Liu and Brosilow \cite{83,84} and Hillestad and Hertzberg, \cite{80} have described a number of tearing algorithms. The main difference in the algorithms being in the way tear streams are estimated. In all the modular algorithms there are essentially two ways of dealing with inaccurate tear stream estimation. These are to either reduce the time horizon used for the next
integration or to reiterate on the current one.

The general algorithm is now described. In the following description \( u_i^t \) is the vector of known tear stream inputs at time \( t \), \( u_i^{t+TH} \) is the vector of known tear stream inputs at time \( t + TH \) and \( u_i^{t+TH}(t) \) is the vector of estimated tear stream variables at time \( t + TH \) calculated by extrapolation from known values up to time \( t \).

For each unit in the system, execute the following concurrently:

1. Predict the tear streams at time \( t + TH \), \( u_i^{t+TH}(t) \), by extrapolation.

2. Solve the local problem \( f_i(\dot{z}_i, z_i, u_i^{t+TH}(t), v_i) = 0 \), using the local integration method. Use an interpolation formula to obtain input values for times between \( t \) and \( t + TH \).

3. Equate the output streams of the unit to the respective inputs of the connected units.

4. Calculate the relative error, \( \varepsilon \), between the calculated and estimated tear variables.

\[
\varepsilon = u_i^{t+TH} - u_i^{t+TH}(t) \tag{3.8}
\]

5. Analyse the error. If unacceptable there are two options: Either reduce the time horizon and continue, or re-estimate the tear streams and return to step 2.

In the latter case the new tear stream estimate can simply be set equal to the latest calculated value. This is essentially direct substitution and leads to a solution method analogous to the Gauss-Jacobi method for linear systems. This is also referred to as Waveform Relaxation by Skjellum [61] and Secchi et al [62,63].
6. Set \( t \) to be \( t + TH \) and update old and new variables.

7. If not finished go to step 1.

The Jacobi approach is guaranteed to converge if all the sub units converge. Similarly Liu and Brosilow [83] state that the algorithm will converge for any problem which can be shown to have a stable unique solution.

### 3.1.3 Comparison of Methods

Overall, modular methods appear to provide more of the desired attributes for parallelism than equation based methods. The following looks at the various requirements and compares the equation based and modular approaches in each one.

- **Robustness**

  Equation based solution is as robust as the solver. Given a good solver, there still remains the problem of discontinuity handling and the requirement to tackle all unit problems in the same manner. Also, including other modelling approaches to given blocks is not possible. Finally equation-based systems have fallen behind on robustness issues in the past. Equation based methods were advantageous mainly from the viewpoint of design and optimisation calculations in flowsheeting. At the dynamic simulation stage the process will already be specified and hence the main advantage given by equation based methods is not required.

  By modularising the problem, each simulator can be assigned a suitable algorithm geared to the local problems characteristics. This will also allow the inclusion of very complex models such as Computational Fluid Dynamic (CFD) models, cellular automata models and the like. Some of these may
also be parallel in their own right under a modular approach. Some of these blocks may also use equation based methods to solve the local problem. This flexibility should enhance robustness to an approach already known for its robust characteristics.

- **Efficiency**

  By allowing different integration methods and different local integration steps across models, each sub block can be optimised. This is not possible in equation based systems where all equations are discretised at the same point. In such an environment the modular approach should be more efficient when compared using the same integration method. Hillestad and Hertzberg [80] present results to this effect.

- **Ease of modelling and reusability of code**

  The very nature of modular methods allows for the simple inclusion of existing simulation code. Similarly providing new models is also simplified. Utilities such as simulation interaction tools can be greatly simplified, since each tool can be tailored for specific unit types and can link directly to the local model rather than the whole system. This applies to simulation output utilities as well.

- **Parallelism**

  The modular approach maps as well to parallelism as the equation based approach. Both methods require some form of global management process which will always produce some sequential part to the overall solution and hence non optimal speedup.

  For the modular approach most of the work can be distributed. The only management required is time horizon and convergence control. Equation based methods require similar management features, but are also likely to require a local linear system to be repeatedly solved. Overall modular methods appear slightly better.
From the above it would appear that a modular approach to simulation provides us with more of the features we require for parallel processing than equation-based methods.

Many researchers are currently looking into the various ways of solving large sets of equations on parallel machines. Conversely for the parallel modular approach there has been very little research. This is especially the case with regards to using very complex models and making these simulators interactive and user friendly enough to be usable by the industrial community.

For these reasons, the modular approach has been used for all of the work presented in this thesis. It would be a brave thing to say that the modular approach is definitely the one to use. The arguments for and against have been going for many years and are likely to do so for the foreseeable future. From the viewpoint of this thesis, equation based methods are not being disregarded, they are merely being left to the mathematicians.

The remainder of this chapter describes how the modular approach can be applied to dynamic distillation simulation. The majority of the theory presented is also applicable to complete plant simulation. The actual implementation of the theory for distillation is described in Chapter 4 and for complete plants in Chapter 5.

3.2 Parallel Dynamic Simulation of Distillation

Chemical processes can easily be broken down into sub blocks by breaking the overall process structure around individual processing units. This leaves us with the problem of simulating these individual unit blocks. Unfortunately some of these unit blocks are themselves computationally expensive to solve, and ideally should be parallelised in their own right.
As with complete processes some sub blocks contain a high degree of modularity at a more conceptual level. Examples include distillation columns, reaction systems and heat exchange networks. It should be possible to apply the same modular approaches as we intend to use for complete processes onto these units.

Based on this theory, it was decided to initially focus on the use of parallelism for the dynamic simulation of a single process unit. The process unit selected was the distillation column. This section deals with the theory of the distillation column and how it can be simulated using the modular approach already described.

### 3.2.1 Distillation

The distillation column was selected as a suitable starting point for exploring the modular approach to dynamic simulation for the following reasons:

- **It is a major piece of equipment.**
  
  The distillation column is one of the most common pieces of equipment used in chemical processing.

- **It is conceptually modular.**
  
  The distillation column can be viewed as a number of vapour/liquid equilibrium stages connected together. The arrangement of the stages is analogous to a chemical process with high interconnectivity between process units.

- **Complex to simulate.**
  
  The equations which represent distillation systems display all of the characteristics displayed by complete chemical process systems. They are stiff, highly nonlinear, are a set of DAEs and contain discontinuities.

- **Difficult to visualise intuitively**
It is very difficult to tell where components accumulate and move during the operation of a real column by simply analysing outlet streams and temperatures. Simulation, especially with good visualisation can provide this information. The ability to visualise this sort of information was key to performing successfully the industrial case studies examined here.

The characteristics of distillation made it an ideal starting point for research. Many of the features envisaged for full process simulation could be built into the initial test package: highly complex models, programmability and visualisation and interaction mechanisms. Starting on a smaller scale also offered an ideal environment for gaining experience with parallel programming, which at the start of this project was extremely difficult because of new hardware systems and the unreliability of early communications software. As it turned out starting small was for the best. Only after many releases of software and a complete language change was the eventual simulation package, called PDist, produced. The implementation and development history of this package is described in the next chapter. The remainder of this section deals with the modularisation approach used.

3.2.2 Modularising Distillation for Parallelism

The distillation column can be viewed as a number of single vapour/liquid equilibrium stages which pass information, in the form of liquid and vapour, to and from their nearest neighbours. Figure 3.3 below shows a conceptual stage.

Although the figure represents a tray process, a packed stage section can be viewed in much the same way, the main difference being the way liquid and vapour are being contacted. Each stage is affected not only by local conditions, but by the stages it is directly connected to. In the following description, the variables associated with the liquid stream will be labeled by the vector $x$, and
those associated with the vapour by the vector $y$. Added to this will be a subscript describing the stage of the column to which the vector refers to. Function vectors are similarly labeled except a subscript is added describing the vector type on a given stage. e.g. $f_{i,j}$ refers to a vector of functions of type $j$ on stage $i$.

Using this notation a typical stage can be expressed by a number of functions of its local and neighbour variables.

\[ f_{i,1}(\dot{x}_i, x_i, x_{i+1}, y_i, y_{i-1}) = 0 \]  

(3.9)

\[ f_{i,2}(x_i, y_i) = 0 \]  

(3.10)

$f_{i,1}$ refers to the mass and heat balance equations across the stage, $f_{i,2}$ refers to the equations used to define liquid and vapour interaction.
The complete set of equations can be represented by:

$$ F(\dot{X}, X, Y) = 0 $$  \hspace{1cm} (3.11)  

The equations are predominantly block diagonal, except in the case where offtakes are reintroduced into the column. This makes the system highly sparse. The larger the number of trays the sparser the system.

The equations represented by equation 3.9 can be integrated by replacing the $\dot{x}_i$ derivatives with finite difference approximations. Depending on the form of the integration formula different effects are obtained. There are essentially two possible forms: explicit and implicit.

(a) – Explicit integration  
(b) – Implicit integration, serial arrangement  
(c) – Implicit integration, parallel arrangement

Figure 3.4: Information Flow for Various Distillation Solution Methods
Explicit Integration

If an explicit or forward difference formula is used then 3.9 can be reduced to the form:

\[ f_{i,1}(x_i, x_i^o, x_{i+1}, y_i^o, y_{i-1}^o) = 0 \]  \hspace{1cm} (3.12)

From this equation set it is possible to calculate \( x_i \) and using equation set 3.10 \( y_i \). This can be done without ever referencing the new time variables of any other block, and hence all blocks may be solved in parallel. The speedup for such an approach should be close to the theoretical maximum. The flow of information for this approach is shown in scheme (a) of figure 3.4 below.

Unfortunately explicit methods are not usually successful, since they are prone to numerical instability or require very small time steps. It is more common to use a fully implicit method.

Implicit Integration

If an implicit or backward difference approximation to \( \dot{x}_i \) is used, equation 3.9 takes the form

\[ f_{i,1}(x_i, x_{i+1}, x_i^o, y_i, y_{i-1}, y_{i-1}^o) = 0 \]  \hspace{1cm} (3.13)

This significantly complicates the local block of equations. Equations 3.13 and 3.10 must now be solved simultaneously and hence iteratively for \( x_i \) and \( y_i \). Blocks cannot easily be run in parallel since each block requires new time information from adjacent blocks, \( x_{i+1} \) and \( y_{i-1} \), which will not be available until
these blocks have themselves been solved. Usually this will result in the need to iterate over all the blocks as well as locally, unless the new connection stream values can be accurately estimated.

The various ways of decoupling the blocks were described in the last section. Given the high interconnection of the blocks it would be extremely difficult to estimate all of the interconnection streams over a given time horizon. Given this we are left with having to use an iterative approach to solving the collective blocks.

On a sequential machine the most efficient method of performing this is to rearrange the equations in the block to allow for a purely sequential calculation path through the column. This is done by rearranging 3.10 to compute $y_i$ from $x_i$:

$$y_i = f(x_i)$$

(3.14)

Given such a relationship 3.13 can be rearranged to compute $x_{i+1}$ from $x_i$, $y_i$ and $y_{i-1}$:

$$x_{i+1} = f(x_i, y_i, y_{i-1}, x_i^0, y_i^0)$$

(3.15)

This is going against the usual philosophy for the sequential modular approach, since the blocks no longer model the actual physical operation of the unit in the sense of calculating outputs from inputs. Scheme (b) of Figure 3.4 shows the flow of information as a result of this method. By starting at the bottom of the column with an estimate of $x_1$, equations 3.14 and 3.15 can be successively solved for each block to calculate $x_2$, $x_3$ and so on, up the column to the top. At the top, block $n$, $x_{n+1}$ should be 0. Based on the difference of $x_n$ from zero a new estimate for $x_1$ can be made. Using this approach we are reducing the problem
to the solution for $x_1$ of the equation:

$$y_n(x_1) - x_{n+1} = 0$$

(3.16)

This approach is efficient in terms of calculation. It is, however, useless from a parallel processing point of view due to its serial nature. It also goes against the input/output model expressed as the desired format. It is mentioned here purely to show the most efficient method for solving the system in a modular fashion on sequential machines.

The final approach available is to solve the blocks for $x_i$ and $y_i$ in terms of $x_{i+1}$ and $y_{i-1}$. The flow of information would be as shown in scheme (c) of Figure 3.4. Such a solution method results in the information flow reflecting the physical process. This time all of the columns streams are being torn at the same time. The various ways of managing these tear streams has already been described in described in Section 3.1.2.

Although likely to be less efficient than the purely sequential method, it does map well to parallelism. The method also has the advantage of allowing individual blocks to be easily grouped together on a single processor and solved as column sections. The main use of this is for when there are less processors available than blocks to be solved. In this case the blocks simply have to be solved sequentially, since only the old iteration variables are used and the blocks are essentially independent of each other. It is possible, however, to use newly updated variables as the block calculations proceed. This changes the local processor algorithm to being more like Gauss-Seidel than Gauss-Jacobi. This results in a Gauss-Seidel approach on each processor within an overall Gauss-Jacobi method. In such circumstances the overall method tends from pure Gauss-Seidel to pure Gauss-Jacobi as processor numbers increase. Although worrying from a stability viewpoint, these results obtained indicate that there is no problem. A similar
method has been used by Skjellum [61] which he calls Waveform Relaxation.

Of these various approaches, the solution methodology built into PDist was to support both the explicit and iterative implicit methods. No attempt was made to implement the multirate approach of tear stream estimation and interpolation for this system. This is catered for in the more recent work on complete process simulation described in Chapter 5. The next section now describes the exact algorithm used by PDist.

3.2.3 Modular Algorithms Used

This section describes the modular algorithms used to solve column models. The algorithm is in two parts: the overall column solution method implemented by PDist and the local column block solution method.

Overall Solution Algorithm

The overall solution algorithm is the Gauss-Jacobi approach with direct substitution. Each processor executes the following functional operations.

1. Load up local stage section setup. i.e number of trays, position, feed parameters etc.

2. Initialise time variables.

3. Initialise the users column section models.

4. Communicate boundary conditions with neighbour column sections. Send subset of $x_i$ and $y_i$ and get $x_{i+1}$ and $y_{i-1}$.

5. Solve local columns section models given the boundary conditions.
6. Test for local convergence. If an explicit integration method is being used, convergence will always be set.

7. If locally converged and all other processor models have converged perform the following:
   (a) Perform any inter time step modelling operations.
   (b) Perform any user interaction, setting the appropriate variables.
   (c) If required store the current solutions.
   (d) Update the time variables.
   (e) If not finished go to step 4, otherwise display the results and stop.

8. Check for excessive iterations. If so display results and stop.

9. Update the interconnection variables and go to step 4.

The column section modelling operations described above are implemented in PDist via a series of interface routines. Each routine is given a specific functionality, and is called at the appropriate time to implement the above algorithm. Everything else is coordinated by the PDist system, including boundary communication, global convergence checking, time management and global interaction and solution management. PDist is analogous to an equation solver for distillation systems which requires a set of operations to be performed by the user.

Local Column Section Algorithm

The local column section solution method is independent of PDist. The column section can be solved as a single block with a single solver or via a modular approach. PDist plays no part in the solution other than to request it, provide the boundary input streams from other processors and to manage the resulting output streams produced.
With the models developed in this thesis, a modular approach is used locally. The algorithm used is as follows:

1. Start at the bottom of the column with current boundary conditions \( y_{i-1} \) provided by PDist.

2. Solve the current stage section, equations 3.13 and 3.10, using an appropriate input/output model.

   If the Gauss-Seidel approach is being used, use the vapour input information newly calculated from the stage below. Otherwise, simply use the input from the previous iteration for this stage. In all models developed by this author, the Gauss-Seidel approach has been used.

3. If at the top of the column return control to PDist. If not move on to the stage directly above and go to step 2

### 3.3 Distillation Models Overview

This section describes in more detail the models which have been written to test PDist. The models have been built around the input/output formulation described above. The actual modelling equations and solution methods are described in Appendix A.

#### 3.3.1 Model Overview

A number of different models have been used within PDist. The majority have been written specifically for this thesis. Other models used were developed by Vasek [85]. Their use with PDist is described by Vasek et al [72].
Two main distillation columns have been modelled: a conventional column and a reactive/azeotropic distillation column industrial case study. For all the models developed, both steady state and dynamic versions have been created. The full industrial case study is described by McKinnel [86].

**Conventional Distillation**

Figure 3.5 below shows the conventional distillation column simulated. The column is composed of a conventional sieve tray section, a conventional reboiler and conventional reflux section with a total condenser and a reflux drum. A number of control options were modelled for the column. The reflux rate can either be fixed by rate, reflux ratio or by a composition controller on the tops product. The boilup can be fixed by rate, boilup ratio or by a composition controller on the bottoms product. Level controllers are used on both the reflux and reboiler holdups, manipulating the tops and bottoms products respectively.

The models can be used to simulate any column of this format, provided that the suitable physical properties are available. Almost all of the models use an implicit, and hence iterative, integration method. The models of Vasek use explicit methods. For physical property estimation, PPDS [87] is accessed via the department's local interface functions [88].

**Reactive Distillation**

As well as conventional column simulation, a reactive/azeotropic distillation industrial case study was carried. The reactive/azeotropic column simulated is shown in Figure 3.6.

The column is designed to carry out a reversible esterification reaction and separ-
rate the desired product from its reactants. The reaction, carried out in a reactor which also acts as a reboiler, can be described as follows:

\[
Acetic Acid + Ethanol \rightarrow Ethyl Acetate + Water
\]

\[
CH_3COOH + C_2H_5OH \rightarrow C_4H_8O_2 + H_2O
\]

The purpose of the column is to remove the product ethyl acetate. To aid sep-
Aration butyl acetate is used as an entrainer. The main feed for the column (E) enters the reactor/reboiler. This feed contains mostly Acetic Acid and Ethanol along with recycled overheads from downstream processing. Acetic acid is in excess in the reactor with only trace amounts leaving in the tops. No bottoms product is taken from the reactor/reboiler. The vapour leaving the top of the column contains predominantly ethyl acetate, water and butyl acetate. Some ethanol may also be present. This vapour stream passes through a total condenser and separates into two layers. Some of the ethyl acetate rich layer is returned as reflux (C). The rest of the two tops products (A) and (B) are then sent to entrainer recovery. The second feed to the column (D), enters on the top tray. This contains a mixture of ethyl acetate and butyl acetate entrainer.
All of the models use an implicit integration method. PPDS was again used for all vapour liquid equilibrium calculations. For the liquid/liquid separator an NRTL model was produced using interaction parameters provided by the company. The reactor was modelled using full reaction kinetics also provided by the company.

The operational nature of the column prevented the use of the steady state models. The column is initially started with a fixed amount of entrainer in the system. This is never allowed to build up once operation has started. The steady state models have no concept of mass holdup. When used, the steady state models produce a solution which contains butyl acetate well in excess of the initial quantity specified, i.e. at its steady state concentration based on the overall mass balance of the column. In order to maintain the entrainer at the correct levels, dynamic models had to be used to calculate the steady state.

The main interest of the company concerned was to be able to actually simulate this column. Attempts at using other packages had been unsuccessful. Once working the main requirements were to study the effects of various step changes and oscillations in the column feeds. In particular the location and movements of the entrainer during operation were of interest since these were unknown and could not be measured on the actual plant. The visualisation provided by PDist was seen as a major feature in analysing such simulation results. The company are also taking an active interest in parallelism and have their own parallel hardware installed.

The models were produced for a very specific problem. However, they have been designed to be general enough to allow their use for other problems. The main requirement is to change the kinetics and NRTL parameters used.
3.4 Summary

The various approaches to parallel dynamic simulation have been presented. Modular based, rather than equation based, methods were selected as the most suitable to use with respect to the overall requirements defined in chapter 1. The subsequent application of the parallel modular approach resulted in an experimental parallel dynamic distillation simulator, PDist. The theory behind the simulation approach used and the various distillation models have been described. The actual modelling equations and solution methods are described in Appendix A.
Chapter 4

PDist: A Parallel Dynamic Distillation Simulator

This chapter describes the implementation of the parallel dynamic distillation simulator PDist. The main emphasis is on the use of parallel hardware, efficiency issues and the provision of the various support tools within a parallel environment. In conjunction with the general implementation issues, the chapter also aims to act as an introductory manual to anyone wishing to use PDist.

4.1 Introduction

PDist was originally designed to be a demonstration program which implemented the parallel simulation theory described in Chapter 3. Through industrial involvement and the need to prove that parallel computers could be utilised in as user friendly a manner as sequential ones, the resulting package is now much more a usable tool than a demonstration.

The majority of software developed for parallel machines ignores the requirement for usability of the software. The main emphasis is on showing how fast the software can run with no excess baggage. For programs such as dynamic simu-
lators, they are useless if the average engineer has to perform a large amount of
adjustments by hand to get things to work. What is required is a package within
which models and data can be manipulated and visualised in a standard way.
The majority of commercial dynamic simulation packages fall short of providing
these types of features.

PDist was developed to demonstrate that a parallel simulation program can be
written which exhibits this desired functionality. Even forgetting parallelism,
PDist is designed to show industry the features which modern software can pro-
vide on top of any simulator and the type of functionality they should be expect-
ing in the commercial packages.

To achieve these aims, PDist is built up from sets of programs. Each set of
programs is designed to perform a specific task. Tasks can be categorised as
either modelling or utility tasks. The programs and their tasks are as follows:

- The Distillation Simulation Programs.

  The simulator is composed of a number of separate programs where each
  simulates a particular section of the distillation column. For all simulations
  there is a reboiler program, a reflux program and one or more stage section
  programs.

  Each simulation program is built from two sets of routines. These are:

  - The Distillation Communications and Solver Routines

    This is the code used to parallelise the simulation calculations across
    multiple processors. The code manages the distillation models, conver-
    gence, interaction and all communications relating to solution transfer
    and boundary condition transfer. Simulation models are linked in via
    a number of modelling interface routines.

  - The Modelling Interface Routines
These routines contain all of the modelling code. The routines each have specific functionality and are called at the correct times by the solver routines.

- The Master Program
  This program controls the setup and execution of the distillation simulator. It is also responsible for the collection and storage of solutions and their graphical display.

- The Interaction Program
  This is a standalone program which can be used to interact with the distillation simulator during execution.

- The PDist Solution Viewer
  This program takes the simulation solutions and displays them in a user defined manner. The code can either be run standalone or called from an existing program. For PDist both options are used with the viewer being linked into the master program for solution display at the end of any simulation. This particular program can be used within any distillation simulator.

- The Runtime Graphics Program
  This program displays simulation solutions during execution.

The programs mentioned cooperate together through a communications layer to provide the overall functionality of PDist. For basic operation there must always be a set of distillation simulation programs and a master program.

The remainder of this chapter is now dedicated to describing the development of PDist and the above functional parts. The categories described are as follows:
• PDist Development History
  This section reviews the overall development of PDist and the various decisions which have guided its development.

• PDist Software and Portability Issues
  This section examines the software and portability issues for PDist and other software developed for this thesis.

• Parallel Distillation Implementation
  This section describes the implementation of the parallel distillation simulator programs and their associated methodologies used to connect to the support tools used.

• The Front End
  This section covers the input, programming and interaction issues associated with the parallel simulator.

• The Back End
  This section deals with the storage of solutions, their management and eventual visualisation.

• Limitations and Future Directions
  This final section describes the limitations of PDist, what can be done about them and what role PDist can play in complex plant simulator.

4.2 PDist Development History

The first work on parallel dynamic distillation simulation began in 1989. All of the software developed at the time was written in a language called OCCAM [6] and was aimed at transputer [89] based MIMD machines. OCCAM was used in preference to other languages purely for reliability and language availability.
reasons. OCCAM is essentially the assembly language for the INMOS Transputer and was the most reliable way of using the local MEIKO hardware at the time. It was not until two years later that robust fortran and C compilers along with usable communications routines began to appear.

The original program progressed as far as simulating a simple binary column along with limited visualisation. In addition to single column simulation some work was also carried out on using the models for predictive control. Two models were run in parallel, one simulating the real column and the other acting as a prediction model for control. The model simulating the real column was made sufficiently different from the prediction model to make the tests more realistic. A description of this and other initial work using the OCCAM program can be found in [69,90]. The early OCCAM work is not further described in this thesis since later work is essentially a recreation and extension of the OCCAM program. The predictive control work referenced represents an investigation into one of the possible applications for fast dynamic simulators.

By 1990, OCCAM had outlived its usefulness. The static and inflexible nature of communications links and the general difficulty in programming and debugging mathematical routines was making further development slow. There was also no way of utilising the existing fortran code available in the industrial community. In particular physical property estimation packages were unusable. Fortunately the first Fortran and C compilers for transputers were becoming available. More importantly a set of communications libraries had been produced which allowed concurrent Fortran and C programs to communicate with each other. This software, produced by MEIKO, was called CSTools [91] and has been the main communications system used by PDist to date. CSTools allows parallel processes to communicate via virtual, rather than hardware, communications links. Virtual communications links allow any processor to talk to any other in the network. The underlying software takes care of routing the messages across the limited
inter processor hardware links. To aid the software routing, the user can wire the processor domain to any pattern desirable within the physical limitations of the connection hardware. For programs, such as PDist, where the most efficient connection strategy is known [92], processes can be placed in an order so as to minimise the number of processors through which any messages must be routed during execution. All this can be performed dynamically from a program and hence means the package can reconfigure itself for different processor numbers without the need for recompilation, extraction and rewiring as with the old OCCAM system. Similar communications software has been written for other hardware platforms. The use of standard languages with linked communications libraries is now the most common method used for programming MIMD machines.

With the advent of CSTools the OCCAM program was scrapped and work began on using Fortran and C for all code development. These standard languages should have allowed for much faster development. Unfortunately, as with the early OCCAM system, the first versions of the compilers and CSTools were bug ridden. Many months were wasted on locating compiler and communications errors. The current versions of this software are now reasonably reliable. Recent work on complete plant simulation has resulted in a move to more portable communications systems. This is and other portability issues are discussed in the next section.

PDist's development has been an evolutionary one. In many ways parallel software development has dictated the pathway the project has taken. As with any new technology this has been, and will be for the near future, in a state of flux. The only way to proceed is to make the best use of what is available until a better option becomes available. Throughout the parallel approach has remained relatively stable. It is only the implementation of the theory which has changed drastically. The decision to move from OCCAM has proved to be a fortunate
one, and although painful at the time has resulted in reduced development times and enhanced functionality of the resulting package. In particular the package could not be run as easily by the average user without the dynamic nature of CSTools' loading and communications libraries.

This concludes the development history of PDist. The next section describes, from a software viewpoint, the current state of PDist and the factors which make and will make it truly portable in the future.

4.3 PDist Software and Portability

The industrial community is sceptical about the use of parallelism on the grounds of hardware dependency and lack of software portability. With this in mind the dependencies of PDist are described and what future developments are likely to provide the portability for PDist and other parallel engineering applications.

The current version of PDist now relies solely on the C and Fortran languages, the X-Windows windowing system for visualisation and CSTools for communications. All of these except CSTools are available on all the major Unix systems in the world. At present PDist will run on MEIKO computing surfaces and single or multiple SUN workstations. The ability to run on single workstations has proven to be of great benefit. With modern multitasking operating systems, programs designed for parallel hardware can run in essentially simulated parallel on a single machine. From an industrial viewpoint this is useful since parallel hardware will not always be available. This was found to be the case with the industrial case study company who used the SUN version when their own MEIKO Computing Surface was in use by other users. The main stumbling block for portability to other platforms is the portability of the communications layer. This problem is now being tackled and already some exciting new software has been produced.
Recently the use of Unix workstation networks as parallel resources has come to the forefront. A number of communications layers capable of running across almost all Unix platforms have appeared. A large proportion of these have been developed in academia and as such are usually in the public domain. The advent of these new communications layers is making the portability problem virtually non existent. The new software is also allowing a heterogeneous approach to utilising computer resources. Workstation networks can now cooperate with specialised hardware, if available, on single problems. With this in mind, recent work on complete plant simulation has resulted in the local development of an independent communications layer RGC (Robust General Communications). This has already been ported to sit on top of CSTools and PVM (Parallel Virtual Machine) [93]. PVM runs on almost all Unix platforms and has been and is being ported to a number of specialised MIMD hardware platforms. Once PDist has been converted to using RGC, all of the software developed for this project will be portable across almost all major platforms. This also includes PCs now that they are of sufficient power to run the Unix operating system. PVM represents a significant player in the move towards producing more portable communication systems.

As well as software, significant developments are taking place in the parallel computer hardware market. The latest trend appears to be towards using very standard processors, in particular those already used in the workstation market. Two of the main vendors, Thinking Machines and MEIKO, have opted to use the SPARC based processors commonly seen in SUN workstations in their latest machines. On top of this their machines are Unix hosted and in the case of MEIKO use a revised SUN Unix kernel on all processors to launch processes and manage communications. Companies like SUN are also producing multiprocessor workstations, similar to those already produced by Sequent. The trend is obvious. Vendors want to provide hardware which is compliant with what is already out there, but yet offer that bit more in the way of processing power. The main
advantage these machines offer is faster communications links. Standard LAN networks do not yet offer high enough bandwidth for the majority of parallel applications. With this in mind the future for parallelism is looking better all the time. In the brief history of PDists development, the package has moved from a transputer based non standard operating system to one which can be ported across the major workstation and parallel hardware available.

Portability of software across hardware will happen, sooner rather than later. For the moment the important task is to highlight the potential areas for exploiting parallelism. Given that portability is no longer a major issue, from this point on, all of the work on PDist is described without reference to particular hardware or software. The text simply assumes the availability of a MIMD environment made up of a number of processors with independent local memory and a modern virtual communications layer.

4.4 Parallel Distillation Implementation

This section describes the implementation of PDist. Firstly an overview of the general decomposition strategy is presented along with the various features which the software aims to provide. This is then followed by an in depth analysis of the modelling interface, the communications harness and finally the complete parallel algorithm.

4.4.1 Decomposition Overview

Figure 4.1 shows the processes, or programs, which make up PDist and the connections between them. These programs are eventually downloaded onto whatever computing resource is available.
In the figure a square box represents a PDist process. The single coloured boxes are utility processes and the twin coloured boxes are modelling processes which are composed of some PDist system code and some user modelling code. In the modelling processes the two code sections are linked via a set of interface routines. The connecting lines represent communication links and the arrows indicate the direction in which information may travel between processes. Where one link appears in the diagram, many software links may be used in the actual implementation.
For basic operation the simulator requires four processes: A reflux process, a minimum of one stage process, a reboiler process and a governing/master process. Together these programs cooperate to implement the parallel solution strategy outlined in Chapter 3. The modelling processes are almost standalone and only require the master process for initialisation, solution storage and solution display purposes. The other processes shown provide interaction and run time graphics support.

As with all MIMD programs, the manner in which the communications between processes is managed is critical to the programs success or failure. In PDist the communications links appear as two distinct types. These are:

- **Solution Critical Communications**
  
  These are communication links along which information essential to solving the simulation problem travels.

- **Utility Communications**
  
  These are the communications links along which non solution critical information travels such as solution output and interaction requests.

A usable simulator requires both sets of communications to work efficiently together. The main aim is to minimise the amount of time spent communicating versus the time spent calculating. The ratio between the two times is very hardware dependent. If communication rates are low, even the most efficiently written program will perform badly. In PDist the majority of work has focused on the efficiency of the solution critical communications. The utility communications are less developed.

Before going on to describe the communication implementation, an explanation of the modelling interface to PDist is given. This is presented here because of the modelling interfaces close relationship to the parallel distillation algorithm.
The two are closely tied, since when PDist is not communicating it is performing some modelling tasks via the interface.

### 4.4.2 PDist Model Interface

The PDist modelling interface is designed to allow PDist to make use of many different distillation models within a single system. The interface is also modularised to provide specific modelling requirements between communications. Figure 4.2 shows the modelling interface.

![PDist Model Interface Diagram](image)

The interface routines are divided into three groups. These groups are each associated with a particular function of the simulator. The routines in each
group are either called by the PDist driver program, or can be used to control the way the driver program operates. The groups are as follows:

- PDist Model Interface

The model interface splits the overall distillation simulation into separate tasks. For every task there is an interface routine. The modeller must provide these routines and include the appropriate modelling code to perform the relevant task. The various routines are called by PDist when required. This enables PDist to hide the majority of the parallelism from the user. Variable selection is left entirely up to the modeller. This enables large pieces of existing code to be used. Input parameters to PDist are provided via global variables at present. This is discussed later. The interface routines and their tasks are as follows:

- pdist_init

  This routine is called at the start of all simulations. Within this routine all PDist initialisation must be performed alongside any model initialisation required by the user models.

- pdist_solve

  This routine is called every time PDist requires the local distillation problem to be solved.

- pdist_converged

  Convergence checking is left to the modeller. This routine is called after each iteration. Within this routine the modeller must check for local convergence, returning true if converged. Once called, PDist works out if all the other processes have converged and takes the appropriate action. For explicit integration methods this function simply needs to return true on every call.

- pdist_changeover
After all the processes have converged, PDist calls this change over routine. Within this routine, all variable updates and once per integration time step changes can be performed.

- pdist_finish

At the end of every simulation this routine is called. It is particularly useful for writing out the status of all models at the termination point. This way it is feasible to startup up future simulations from where they have left off. It is difficult to cater for such things directly in PDist.

- PDist Packing Interface

The packing interface represents the only aspect of parallelism which the modeller must understand to use the simulator. These routines are designed to allow the modeller to pack and unpack the liquid and vapour streams which are to be passed between different simulator modules. Again these routines are called by the PDist driver. The routines are as follows:

- pdist_pack_streams

This is called every time PDist requires the modeller to pack up the liquid and vapour streams leaving the local column section. The stream size is user defined using the call pdist_set_stream_size. See below.

- pdist_unpack_streams

This is called every time PDist receives new stream information from local simulation processes. Since the modeller packs up the streams, the modeller must unpack them as well.

- pdist_solutions

PDist manages the collection and display of all solutions. The structure of the information leaving PDist is defined by the modeller. The mechanism for this is discussed later. This routine is called when PDist wants to pack up and send solutions in the format which the modeller has specified. This makes the system very flexible.
• PDist Control Interface

The control interface routines differ from the other routines in that they can be used within the others to control the way in which the PDist driver code operates. Some of these routines must be used before PDist can function correctly. The routines are as follows:

- pdist.set.mode
  This routine sets the simulation mode. Two modes are provided: steady state mode and dynamic mode. Either can be set as the starting mode. In the case of steady state mode, a dynamic simulation may be run afterwards.

- pdist.set.broadcast.size
  This routine is used to control the amount of shared variable space between processes. Shared memory is implemented to allow processes which are not adjacent to broadcast information to each other. To maximise efficiency the amount of shared space must be declared. An example of this is described later. The shared variables are updated every iteration in a similar manner to the stream boundary conditions.

- pdist.set.broadcast.slot
  PDist provides a block of shared memory for storing double precision numbers. Each number is assigned to a particular slot of the memory. This routine is used to set numerical slots in the shared memory area.

- pdist.set.stream.size
  This routine must be called by the user in the pdist.init routine. It is used to set the size of the streams being passed between modelling processes.

- pdist.set.next.state
  This routine is used to invoke changes pre-programmed into the simulator via the input system. This is described in greater detail later.
The clean termination of parallel programs is always difficult. This routine indicates to PDist that premature termination is required. With this mechanism PDist can always shut down cleanly and make sure that no solutions are lost.

The interface tries as much as possible to hide any parallelism. The breakdown of the modelling into the tasks described is not radically different from the standard approach used for sequential algorithms. In fact it probably forces a more structured approach to model development. The interface is by no means complete. The current set is designed to provide a working and usable demonstration. Many niceties could be added. This includes a more generic input mechanism, rather than the static one currently used. However such additions are also useful in the much wider context of complete process simulation, where other tools will require similar functionality. This and other extensions are described, where appropriate, in the sections which follow.

Now that the model interface has been presented, the overall parallel algorithm and communication structure can be explained. In the text which follows the model interface will be referred to as a means of showing what functions PDist is performing during non communication periods.

4.4.3 Simulation Critical Communications

The simulation critical communications are those communications required to implement the parallel distillation simulation alone. They do not include any communications to provide solution or interaction support. The tasks which these communications must cater for are as follows:

- Boundary condition transfer
CHAPTER 4. PDIST

The passing of liquid and vapour streams between concurrent simulation processes.

- Simulation control

The control of PDist relies on information about the status of all the simulation blocks being available to the decision making process, wherever it may be. It also relies on the decision making process being able to instruct all simulation processes on the various courses of action available.

- Shared Memory Support

In PDist most information is transferred between adjacent simulation blocks. In the event of a process requiring information from farther away, a mechanism must be provided by which one process can access information in another. The simplest way of achieving this is to provide a block of shared memory to and from which all processes can read and write.

- Utility Management

Although not critical to the solution, the communications described here must be designed with a knowledge of how utility requirements will fit into the system.

The bulk of information transferred between processes is boundary condition information. Shared memory information may also contribute significantly to information size, but is highly variable. The information associated with control function and utility management is small in comparison. The main problem is how to combine both types of message efficiently within a single environment.

An efficient parallel program is one which minimises the amount of time spent in a given run communicating and maximises the amount of concurrent calculation. The theory associated with the overall breakdown of the calculation has been dealt with in Chapter 3. There are two sets of factors which affect communication
efficiency: those associated with single communications and those associated with multiple communications.

Single Communication Efficiency Factors

The time taken for a single communication between two processors is dependent on:

- The time taken to set the communication up

Setting up a communication incurs an overhead. This is both hardware and software dependent. The size of this overhead can be significant for small messages.

- The size of the message being communicated

![Diagram showing communication time vs packet size on MEIKO CS1](image-url)
Figure 4.4: Effect of Various Factors on Communication Efficiency

The larger the message, the longer it takes to communicate it. Transmission rate is usually a fixed quantity and hence time is a linear function of size.

- The route the communication path takes

  The route a message takes significantly affects the time taken. Where two processes are not directly connected, a communication may be routed through another processor. For every routing, the equivalent of another complete communication is being performed. This is again highly hardware dependent.

The above all significantly contribute to the overall time for a communication. The graphs in Figures 4.3 and 4.4 illustrate these factors using communications timings from a MEIKO Computing Surface and a SUN SPARCstation respectively. The graphs show the average time taken to pass packets of varying size between two processes which are either directly connected or farther apart requi-
ing messages to be routed through other processors. In the case of the SPARC-station the link is either virtual on a single machine or across the ethernet.

From the graphs one can see that, even for directly connected processes, the overhead associated with setting up a communication is significant. When sending small packets it is better to tag them onto existing messages, when possible, than to pass them in a separate communication. In PDist there are the large boundary condition packets and the small control related packets. It is more efficient to pack all of these together rather than have separate messages. This is especially the case for connected workstations where the communication startup overhead is much more significant. From Figure 4.3 it is also obvious that routing can affect message transfer: not only with respect to startup cost, but transfer rate as well. It is thus important to try and arrange for communications to be as direct as possible.

Multiple Communication Efficiency Factors

As well as implementing single communications efficiently, it is equally important to make sure that all communications work efficiently together. Factors which affect multiple message efficiency are:

- Concurrency of communications

  Where possible communications should be passed concurrently. Communication works on the same principle as parallelising the workload. The more you do at the same time, the less time it will take.

- Load Balanced Processes

  Load balancing greatly affects a parallel programs ability to synchronise successfully its communications. If all programs reach their communication point at different times some will undoubtedly end up waiting for others
to catch up before being able to communicate with them. This problem becomes severe with tightly coupled programs like PDist.

The two factors are very closely related. Concurrency of communication can only occur if all programs are able to communicate at the same time. Load balancing a program is highly complex. In distillation load balancing can be roughly achieved by placing equal numbers of trays in every modelling program. Unfortunately, changes in a column usually start in one location and move around. For this reason the simulation of some trays will always be computationally more expensive than others. This inevitably results in an imbalance. There is no easy solution to the problem. In severe cases it is sometimes worthwhile to move part of the work load from one processor and put it on another. This is referred to as process migration. For PDist the imbalances produced do not warrant such drastic action. However in a complete plant simulator such action may be required.

The Implementation of Solution Critical Communication

The implementation of PDist’s communication layer attempts to incorporate all of the efficiency issues described. Three possible configurations were examined. These are shown in Figure 4.5. The main difference between the configurations is in how boundary, control and shared memory information is managed. The configurations shown have the following characteristics:

1. Centralised Control and Communications

   Each simulator is managed by a central coordinating program. This is responsible for managing all control, shared and boundary information. It is essentially the master/slave approach.

2. Centralised Control and Distributed Communications
CHAPTER 4. PDIST

This is an adaptation of 1. Instead of having a central program coordinating all information transfer, the control and shared information transfer is separated from that associated with boundary condition transfer.

3. Distributed Control and Communication

In this case the control module is built into the simulation programs. Again one process is responsible for final decision making, but the overall message transfer involves combining messages of different types.

It would be possible to build functional simulators from all three of these approaches. Each one provides the overall structure required to implement the functionality PDist requires. The pros and cons of each structure are now examined.

Structure 1

This suffers from having to manage all connection information sequentially in a single program. Given the volume of information being managed, the simulation processes would be idle for a significant period. From a commu-
nication viewpoint there is the benefit of being able to bundle information together. Unfortunately there is no scope for passing these packets concurrently since they are all heading towards a single program which can only receive them sequentially. Likewise the return packets can only be sent sequentially. Overall for \((n)\) simulation processes there are \(n\) messages to the controller and \((n)\) messages returned. Since these are all sequential the communications take \((2n)\) communication time units. There is also a delay associated with the control action taken by the program.

Structure 2

This suffers in a similar manner to 1 but offers more from an efficiency viewpoint. Again there is an extra program for control and there is still the same volume of communication to and from it. This time however the controller has less to do and will thus take less time. Also during the control period the simulators do not need to be idle since they are free to pass their boundary information while waiting. This boundary condition transfer can also be performed in a highly concurrent way. There are \((2n)\) control messages and \((2n - 2)\) boundary condition messages. The boundary condition messages can be passed concurrently in approximately \((4)\) time units, two sends and two receives for every process. Since this can be done while the controller is collating information the overall communication time is roughly \((2n)\) communication time units. The control delay should be less in this case, but with imbalances in loading it is difficult to say whether this or structure 1 would be the most efficient.

Structure 3

This structure immediately offers the benefit of requiring one less program. The implementation is however more involved. The problem is how to pass the boundary information between nearest neighbour processes and yet make sure that the controller process receives information about the state of the whole network.
Before communication every process has boundary conditions to transmit and receive. During this transmission, tagged control and shared memory information must reach the control process. After the control process has finished, the control action and unified shared variable map must be returned to every process.

The most efficient method for achieving this involves leapfrogging control information from one boundary condition packet to the other during the boundary condition communication phase. To make this possible the boundary conditions must be passed in a sequential manner to allow the information to be passed on from one message to the other before transmission. To do this the control process must first be placed in the centre of the column pipeline. Then, starting from either end the tops and bottoms processes pass their liquid and vapour boundary conditions respectively to their nearest neighbour processes. Each boundary condition packet contains the convergence status and shared memory slots associated with the sending process. The neighbouring processes then receive these packets, remove the control related information and add it to their own boundary condition packets. These are then sent on as before. This results in a wave of information heading from the outside to the centre. At each passing the process can AND its convergence state with that just received. Similarly the local shared memory information can be set. Eventually the central control process receives two packets of information: one from above and one from below. The control information can be removed from these packets and combined with the control process state to provide a unified view of all the processes in the network. At this point the control process can decide on the next course of action. The reverse communication sequence can now be initiated. This time the other half of the boundary condition information is transmitted, starting at the control process and moving outwards. Each boundary condition packet now includes the new control information and the now unified shared memory map. As this returning wave passes each
process, the process can move on to begin whatever task has been set by the control process.

For \( n \) processes there are \((n - 1)\) inward communications. Communications on either side of the control process can be performed concurrently. The control process can only receive the above and below incoming packets sequentially which results in the inward wave of packets taking a maximum of \((n + 1)/2\) communication time units. During this communication transfer many of the control decisions have already been made. By the time the control process receives the inward packets, the only actions required are to AND the local convergence state received with those of above and below and to combine the shared memory slots together. This takes very little time. The outward wave of information similarly takes \((n + 1)/2\) communication time units. Overall the time taken is \((n + 1)\) time units with negligible control overhead. This is approximately twice as fast as the time taken for the other structural options.

Finally this approach also scales better. With the master slave approach in 1 and 2 many processes are writing to one. In such cases routing is likely to play a significant part in message transfer times as process numbers increase. Message transfer is always nearest neighbour with this final structure and so routing should never present a problem.

From the above we can see that structure 3 is the most efficient. This is the structure which has been used in PDist. This particular structure only owes its existence to the fact that the movement of boundary conditions allows information to travel in both directions. This is very particular to distillation. For a complete plant simulator the direction of boundary information is predominantly uni-directional, and hence a structure similar to 2 would be more efficient.

So far the actual control methodology has been ignored. It has only been described with reference to packet size. The next section describes what the control
information actually contains.

Simulation Control Structure

The control information is divided in two: that passed with the inward wave and that passed with the outward wave. The inward wave contains information about the processes passed and this includes:

- **Convergence state**
  
  This is either true or false and gives the convergence state of the column from the current process to the farthest out.

- **Overall Current State**
  
  Preprogrammed setup changes are executed either at a given time or when the simulation reaches a particular state. At any time the overall simulations state is represented by a number. When a particular state has been reached in the simulation, the process responsible for identifying when this point has been reached can increment the current state level using the modelling call `pdist_set_next_state()`. The inward wave is used to inform the control process of the highest state reached. This is then broadcast to all process on the outward wave.

- **User Control Requests**
  
  Through the wave, processes can indicate special requests from the control process. An example is a user shutdown request which would tell the controller to cleanly shutdown the parallel application and save the results.

- **Time Step Changes**
  
  Through the wave mechanism it is possible to present the controller with the minimum requested local time step from all the processes. From this a
global time step can be set by the controller on the outward wave.

- Shared Memory Information

As the wave passes each process it sets its local shared variables. By the time the controller gets the inward waves it simply has to combine the two blocks with its own, thus producing a unified block ready for retransmission on the outward wave.

The outward wave contains the control actions:

- The next simulation action to be performed:

  **Shutdown** Cleanly shut down.

  **Solve** Continue on the current time step for another iteration.

  **Changeover** Start next time step.

  **Changeover and Interact** Allows the controller to tell all the processes to go into interaction mode after the time step changeover has been completed. Interaction can only occur after a time step has been completed.

- Overall simulation state

  The highest state reached by the simulation. Once received each process must execute any, state labelled, preprogrammed setup changes.

- Time step information

  The next time step to be used.

- Shared memory data

  A complete copy of the shared memory map
This control structure is simple, extensible and above all reliable. Interaction utilities are easily catered for since the control process can switch the other processes into different modes. If extra functionality is required, extra modes can simply be added.

4.4.4 Utility Communications

The utility communications must blend as efficiently as possible with the solution critical ones. At present the utilities supported are simulation interaction, solution management and run time graphics.

Interaction Handling

The support for interaction handling is relatively simple. The mechanism essentially involves sending changes in the original column setup to each of the simulation processes being used.

Interaction with the simulator is an infrequent event. For this reason the interaction system should be relatively non-intrusive on the normal function of the simulator as a whole. The other main issue is how interaction should be handled. There are two options:

1. The interrupt/change/restart approach

   With this approach a change to the simulation setup can only be made by first interrupting the simulation. The simulation would then be in interactive mode during which period the desired changes could be made. Finally the simulation is restarted.

2. The online change approach
Here the data is changed as soon as it is registered without the need to interrupt the simulation.

The implementation of both cases involves executing the steps laid out in option 1. The difference is in who is in control of the sequence. At present option 1 has been implemented. This is essentially down to the behaviour of the interaction graphical user interface (GUI). Option 2 could easily be implemented as an alternative.

Either way the mechanism for actually making a change is the same. The simulation processes must be told to go into interaction mode followed by a restart message. Within the current control communication structure the control process uses the outward wave of information to inform all processes about the next course of action. The best approach is to inform the control process that interaction is required and to get it to inform the other processes that they should enter interaction mode.

Interaction has been implemented in this way. After every time step has completed, the control process in PDist looks to see if the interaction panel is trying to communicate. This involves using what is called a non-blocked receive. This type of receive either returns nothing or a message and is very fast to execute. Using this mechanism the control process is unaffected by the presence of an interaction panel until the panel registers an interest in communicating. Similarly the interaction panel only uses up resources when it is being used. The rest of the time it is idle.

Once the interaction panel has made contact with the control process, the simulation can be shut down by passing an interaction instruction with the outward wave. Each process then goes into interaction mode which essentially involves receiving setup changes and processing them. The final action is to send the control process a restart signal, after which the simulation simply continues as
before. The actual message structure used for interaction is described in Section 4.5.

As well as interaction via an interaction panel, PDist also allows changes to be preprogrammed before execution. These changes are loaded onto the individual processes at run time and can easily be executed at the end of the appropriate time steps. Again this mechanism is explained in Section 4.5.

Overall the interaction mechanism is efficient, non-intrusive and fits easily into the existing communication structure. It can easily be switched off by simply not running the interaction panel program. Such functionality shows the benefits of breaking a monolithic application up into smaller and more focused communicating ones.

Solution Handling

The collection of solutions from the simulator presents a much larger efficiency problem than interaction. Here information must be exported from the simulator at regular intervals to a central collection program. As with simulation control there are a number of ways this can be done.

The most obvious method is the master/slave approach. This involves a central collection program, or master, receiving solution packets from each simulation process. This has already been shown to be inefficient since the collection process can only receive one packet at a time. However, due to its ease of implementation, this method has been used in the initial version of PDist. The overall time spent sending solutions is only significant where simple distillation models are used. As the timings results in Chapter 6 show, for more complex models, the transfer of solution information is small in comparison to the calculation time for the models used.
Although the master/slave approach has been used, another option is available. Within the existing simulator the inward wave of boundary data has already been used to pass a global view of the shared memory data for all processes. This mechanism could also be used to pass the solutions for all processes to the central control process. Once this has obtained all of the solution data it could transmit it as a single solution packet out of the simulator. This is likely to be more efficient than the master/slave approach for the same reasons as those given when considering the solution critical communication structures possible.

Run Time Graphics

The run time graphics present much less of a problem since the program generating the graphs simply requires a copy of all solutions given to the master process. In fact, the amount of data passed can be reduced to the amount which is actually to be displayed. This can also be passed within a single packet. At present all data entering the master process is simply copied and sent on to the runtime graphics program. This program was originally created to show a concept and has not been the main focus of attention throughout the project.

4.4.5 Complete Parallel Algorithm

The complete parallel algorithm, including model interface calls, is now described. The algorithm is presented using pseudo code. The code description includes the action for the reflux, stage and reboiler processes. In PDist at present the reflux and reboiler processes are distinctly separate. In any future version a single program would be used for all process types.
Run the procedure PDistMainProgram in parallel with itself.

procedure PDistMainProgram
Initialise local PDist variables and the communications layer
Analyse local configuration, i.e., Number of plates, Interactive status etc...
Create the communications links
Register ports to receive incoming stream information
Locate external ports to transmit outgoing stream information to
Locate port on master to send solutions to
if interactive and the control process then
  Register a port to receive interaction information on
  Locate the port on the interaction process to send back replies
end if
Download the initial column setup from the master process
Extract the PDist simulation data from the setup and initialise local variables
Extract the PDist model information and set the user setup variables
Initialise the PDist models by a call to pdist’init()
Analyse the simulation variables setup within pdist’init() via the PDist Control Interface

simulating ← true
while simulating do
  Pack up the outward boundary streams by a call to pdist’pack’streams()
  Perform the boundary condition wave communications:
  call procedure PerformBoundaryCommunication()
  Perform the requested control action set in outward wave communication:
  call procedure PerformNextSimulationOperation()
end while
Simulation has completed. Finish up with a call to pdist’finish()
Close down the communications layer

procedure PDistMainProgram
procedure PerformBoundaryCommunication
  if you are the reflux process then
    Tag convergence, state and shared variable info onto liquid out stream
    Start the top inward wave off by sending the liquid stream
    Receive the outward wave vapour stream
    Unpack the control signal, state and shared memory block
  else if you are the reboiler process then
    Tag convergence, state and shared variable info onto vapour out stream
    Start the bottom inward wave off by sending the vapour stream
    Receive the outward wave liquid stream
    Unpack the control signal, state and shared memory block
  else if you are above the control process then
    Receive the incoming wave liquid stream and remove tagged information
    AND received convergence state with local one
    Add local shared memory slots to those received
    Set the simulation state to maximum of the received and local state
    Tag new information on to liquid out stream
    Send the liquid stream to the inward process
    Receive outward wave vapour stream
    Copy tagged information onto the outward vapour stream and send it
  else if you are below the control process then
    Receive the incoming wave vapour stream
    AND received convergence state with local one
    Add local shared memory slots to those received
    Set the simulation state to maximum of the received and local state
    Tag new information on to vapour out stream
    Send the vapour stream to the outward process
    Receive outward wave liquid stream
    Copy tagged information onto the outward liquid stream and send it
  else if you are the control process then
    Receive incoming liquid stream from above
    Receive incoming vapour stream from below
    Extract convergence and other tagged information
    Check for global convergence
    Set the outward control signal:
      if global convergence and interaction is waiting then
        set the control signal to INTERACTCHANGEOVER
      else if global convergence then
        set the control signal to CHANGEOVER
      else if user terminate signal set in inward wave then
        set the control signal to USERTERMINATE
      else if the end of the simulation then
        set the control signal to TERMINATE
      else
        set the control signal to SOLVE
    end if
    Combine the received shared memory slots with the local ones to create a unified block for export to all processes on the outward wave
    Set the global State level based on maximum of those received
    Tag the new control information onto the outward liquid and vapour streams
    Start off the outward wave by sending the liquid and vapour streams
  else
    simulating ← false
  end if
end procedure PerformBoundaryCommunication
procedure PerformNextSimulationOperation
  if action is INTERACT CHANGEOVER then
    Send solutions to master if applicable. Pack using pdist'solutions()
    Execute any preprogrammed setup changes
    Enter interaction mode:
      if you are the control process then
        Send a confirmation packet to the interaction process
        interacting ← true
        while interacting do
          Receive an encoded setup change
          Send copies to processes above and below
          Decode the setup change and set the local setup variables
          if setup change is a finish then
            interacting ← false
          end if
        end while
      else
        interacting ← true
        while interacting do
          Receive encoded setup change from inward process
          Send encoded setup on to the outward process
          Decode the setup change and change local setup variables
          if setup change is a finish then
            interacting ← false
          end if
        end while
      end if
    Perform end of time step updates by a call to pdist_changeover()
    Check simulation time against the finish time Set TERMINATE if done
  else if action is CHANGEOVER then
    Send solutions to master if applicable. Pack using pdist'solutions()
    Execute any preprogrammed setup changes
    Perform end of time step updates by a call to pdist_changeover()
    Check simulation time against the finish time. Set TERMINATE if done
  else if action is USER_TERMINATE or TERMINATE then
    Send solutions to master if applicable Pack using pdist'solutions()
    simulating ← false
  else if action is SOLVE then
    Unpack the received boundary information with pdist_unpack Streams()
    Perform any preprogrammed state changes if the state has changed
    Solve the local distillation models by a call to pdist_solve()
    Find out if models have converged by a call to pdist_converged()
    if you are the control process then
      if interaction message present then
        set the interaction pending flag
    end if
  end if
end procedure PerformNextSimulationOperation
4.4.6 Implementation Summary

Overall the majority of attention has been focused on getting the solution critical communications to be efficient. There is little which can now be done to enhance the solution critical communications structure which has been implemented. As for utility management, the current communication structure is less well defined. Some improvements to the solution and runtime management could be made. However, the current utility communications represent an almost worst case scenario. Given that at present they do not present any major efficiency problems, any improvement is likely to make the utility management relatively insignificant. A full analysis of the various communication times during execution are given in Chapter 6.

The rest of this chapter now focuses on the utility programs written for PDist and how they represent demonstrations of the type of utilities that could be used in a full dynamic plant simulator.

4.5 The Front End

The front end of PDist allows the user to setup and interact with the distillation models being used. This section describes the overall simulation input problem and describes the input mechanism which has been used for PDist.

4.5.1 The Dynamic Simulation Input Problem

Creating a generic input format for any dynamic process simulator is difficult. Any format created must be capable of describing the following:

1. The process to simulate. This is composed of two parts:
(a) The process structure and associated process unit attributes
(b) The specific attributes required by the models used to simulate the process units in question

2. The changes to the above structure and attributes during a simulation.

In any simulation, the original process setup changes as a function of time. The changes can be either modelling changes, structural changes or both combined. Figure 4.6 highlights such changes for a distillation process with time. Describing these changes on paper is relatively simple and obvious when viewed by the trained eye. Providing a similar mechanism for a program to analyse is a much greater problem.

Describing a Process

A chemical process is composed of two obvious parts: The actual chemical processing equipment and the control mechanisms which have been added. The control layout depends on the equipment available and hence the actual equipment layout must first be defined.

A chemical process is built out of connected pieces of process equipment. Every piece of equipment can be categorised as being of a specific type. Also a given type of equipment may be made out of other types of equipment. For every type of equipment there is a specific set of attributes. An example is a sieve tray column which always has a number of trays as an attribute but is built out of a number of tray equipment types. Within these tray equipment types would be all of the attributes associated with a tray, i.e. weir height. Given a standard representation for different pieces of possible equipment it is relatively simple to see how a simulation model could dissect a structure to get the physical attributes required.
Given that the chemical process structure has been defined, there is now the problem of control information. Before attaching a control system, there must first be points for the controllers to attach to. Example points are analysis ports of varying types, valves and the like. For a process built out of well defined equipment, the equipment definition must cater for the possible locations where control ports may be added. For example a tray equipment specification should allow for the possibility of a number of temperature and pressure probes being
present. For any single process structure there is a large variety of configurations in which control information can be added. Also in a given process more than one control system may be present on a given set of equipment, some active and some inactive depending on the actual operational mode of the process.

We are now left with a highly complex process description, which like a real process contains many options of structure, depending on how it is being used. Given that a storage format can be produced which represents all this information and likewise a nice process drawing package on top to simplify its creation from the user, how would a modelling program go about analysing this process and whether it is capable of actually simulating the structure described?

Modelling Based on a Process Description

Simulation models are usually written with a specific process structure in mind. For different structures, different models are written. This includes both control and equipment models. The problem for a given set of models is in how to analyse a given process and work out whether the models available apply to the structure provided. There are a number of possible ways of tackling the overall problem. The most obvious ones are:

1. The modeller produces a series of possible input process templates.

   In this case the modeller would advertise the possible process structures that can be simulated. The front end of the package could then compare all input processes presented with the templates provided by the modeller and decide which if any of the models are applicable.

   Once selected the models would then simply interrogate the process structure using some dissection functions provided with the process description library.
2. The modeller writes a process analyser for the particular models.

Here the modeller writes a process analyser using function calls from the process description library. Using this mechanism the models are free to explore the process rather than simply having to take one which matches a particular template. The idea is similar to the template option except here the templates are represented in the programming rather than as input descriptions in their own right.

Option 1 offers a more generic method of describing possible input. For this mechanism, a front end process matcher would be usable for many simulators of varying types. The main problem is that at first glance, the format is very static. A template would have to be provided for all possible variations around a theme. What would be much more usable would be a template description which has a more dynamic flavour: i.e. one that describes the theme and the rules under which it is applicable. An example would be a controller that requires a temperature reading from the stripping section of a column. This temperature reading could thus be taken from any tray under the feed tray. Since the number of trays and the feed location are not known until a simulation is requested and an input object provided, the template for the models should be able to state rules relating to actual parameters in the input object. In our example this could be a rule stating that the control connection must be to a probe which is at a tray number below the feed tray in any column used. Another type of rule which the template could contain is one relating to limits of the models usage for a given problem type. An example of this type of rule is that a particular model cannot cope with more than a certain number of components. A simulator should never be run with an input it cannot cope with.

Option 2 is unusable as a generic tool, since the modeller is being asked to write a specific process analyser. In this case the models are also up and running when the input process is received. The advantage of this option however, is that it
does provide the modeller with a more dynamic viewpoint of the simulation input. Explicit knowledge of the limits of the program etc can be hard coded into the analyser.

Of the two, option 1 is by far the most attractive but would require much more work to develop. This option not only requires a method for describing a process structure, but also a layer above which can explore and validate particular parts and relationships within the process structure itself. Unfortunately this particular option represents the only real solution for a large process simulator which is built out of smaller more focused ones. At some point the front end of such a simulator has to break the overall problem up into parts. The only way it can do this is by knowing what simulators are available, and what types of inputs they can accept. The overall problem then boils down to taking the overall process, separating it into a number of simulatable parts based on knowledge of the simulators available and putting them back together. Only a simulator with this functionality is really going to be simple to use for large problems. The theory behind this type of input is further explored in Chapter 5.

In addition to the general process structure, the specific information associated with a given modelling methodology must be present. In the template described here, this would have to be incorporated as a set of rules which define the information required in addition to that obtainable from the basic process description. How this information would be input is a difficult problem in its own right.

Finally whatever structure is defined, it must also be able to represent changes that occur with time. This can either be done by providing a list of time stamped formats, or as a single format with a list of time stamped changes. Either way, the changes must be valid as far as the models are concerned and also easily analysed by the modelling code.
Input Format Conclusions and PDists Input

The input format described represents the ideal format for a dynamic simulator of any type and size. The work required to produce such a mechanism is enormous. It may not even be possible. The programs produced in this thesis aim to show how a simulator can be broken down into parts. It is thus also part of this thesis to attempt to show how an input structure conceptually similar, but much more simplistic, can be used to input structural and control related information in a standard way to a simulator which can technically contain many different models.

Many of the complex issues described in this section are now being tackled by the épée project [94], of which this author is a part. This project aims to create an environment where all engineering programs used, communicate information to each other via engineering objects. Part of this involves the creation and manipulation of process objects via various tools.

4.5.2 PDist Input Format

The input mechanism for PDist uses a standardised file format to describe a distillation column. Instead of representing the distillation process as a series of connected process units, it uses a more general engineering viewpoint of what an actual distillation column is like.

In the format a column is assumed to be made up from a tray column section, a reflux section and a reboiler section. The file format is structured to allow these various parts to be described. The format is very fixed and designed to be usable for tray columns only at present. If required, it could be tailored for packed columns. The exact syntax of the input format is described in Appendix E.2 and an example is given in Appendix C. The overall format is composed of three sections. These are:
1. The column section and its standard attributes.

2. The reflux and reboiler sections.

3. A programming section for registering dynamic changes to the above.

Column Section Attributes

The column section description is subdivided into contexts. Each context contains the information associated with that context. At the moment the contexts provided are:

- Size information
  The number of components, stages and feeds etc.
- Component names
- Feed information
  This includes location of, composition of, temperature, pressure and q value.
- Thermodynamic information
  Molecular weights, densities etc.
- Simulation information
  Finish time, time step, history time step etc.
- Tray hydraulic information
- Model specific information
  This allows model specific input variable declarations to be made.
All of this information is reasonably standard and unlikely to change between different column simulations. The only really changeable part is the user modelling variables.

**Reflux and Reboiler Attributes**

The reflux and reboiler sections are not so invariant. Probably the greatest variation between columns is in what the ends look like. To work around this problem, the input format for these sections allows models of varying types to be associated with the four main controllable streams. These being the reflux, tops, reboil and bottoms streams. For each stream a number of models or controllers can be defined. To each one the associated model and structure parameters are added. Additionally the description includes a default model to use for each stream. Changes to these defaults can be detected during model execution. This is described in section 4.5.3.

Using this input mechanism it is possible to provide a single input format to a set of models which are capable of simulating many column structures. The industrial case study model input file shown in Appendix C displays this functionality. Here the input describes models capable of simulating both conventional distillation columns with various control options and also reactive azeotropic distillation columns. The choice of structure is controlled by changing the default model selection variables mentioned above.

**Programming Attribute Changes**

The final part of the input mechanism allows a number of preprogrammed perturbations to the initial column setup to be entered. These changes are referred to as events in PDist. There are two types:
1. **TIME events**

These events are set to execute at a particular time in the simulation.

2. **STATE events**

A STATE event differs from a TIME event in that it is executed when a simulated column reaches a given state of operation. The exact point at which this occurs can only be detected by the simulation models themselves. Each STATE that is likely to occur is given a preprogrammed number. When a set of models reach the expected state, they can execute all of the events registered with this state. An example use of this would be for declaring events to execute when a dynamic simulation reaches steady state.

These changes can either be preprogrammed, added dynamically via the interaction panel or a combination of both. Either way this mechanism allows a complete log of all changes to be kept. After an interactive session the original input file is reproduced, complete with additional TIME events added. When PDist is run in batch mode with this produced input file, it will completely recreate the simulation performed interactively.

This recording structure would be most useful for operator training. A simulation could be set up and preprogrammed with known changes. The trainee could then run the simulation interactively and try to correct the hidden changes based on output observations from the simulation. At the end of the simulation the revised input file could be used to review the changes made. If wrong actions were taken, the file could be adjusted and the simulation rerun to show the correct course of action.
Input Format Summary

The input format described provides a mechanism for describing a column, if somewhat abstract, and the changes which must be made. It can also be used by a variety of tools, which understand the format, to pass column setup information between each other. All of the tools in PDist read the input file on running. In interactive runs, the interaction panel uses the format to inform other tools of new setups.

4.5.3 Accessing Input Information from PDist Models

At the beginning of a simulation the input format is parsed and the associated information bundled up and communicated to the individual model simulators which make up PDist. The communicated information is then unbundled and presented to the models as global setup variables. The variables used are shown in Appendix C.1.

The models can either use these variables directly or copy over the contents to local storage. As well as the initial setups, the individual simulators keep track of any events which are preprogrammed. The input variables are updated at the appropriate times. Any models which directly link to these variables, will automatically pick up any changes. Changes made via the interaction panel are encoded and transmitted to the various simulators for decoding. Both preprogrammed and interactive changes are always presented to the models via changes in the global setup variables.

This mechanism, although usable, lacks any real error control. The current mechanism requires the models to check that the input data matches the expected format, spot dynamic changes in the input and cope with illegal model changes. There is also no unit checking for input variables. All inputs variables are simply
numbers. A more acceptable mechanism would be one which provides a layer between the input data and the modelling variables. Providing such a mechanism is again complex. For the purposes of this thesis it would only really offer a more refined way of providing the input functionality which already exists. It has thus not been improved farther. Again the épée [94] project tackles many of the issues discussed.

4.5.4 Interaction

The final part of the front end is the interaction panel. This is a program which can read the input format, allow it to be changed, start the simulation and change the initial setup dynamically during execution.

Figure 4.7 shows a screen dump of the interaction panel during execution. The interaction panel simply displays all of the information in the input format and allows it to be changed. Much of the input format contains information which is purely there to allow the interaction program to present the input in the users own terminology. Specialised windows are provided to allow the reflux and reboiler models to be switched on and off. More appreciation of it is gained from its use rather than its description. The graphical user interface allows full control of the simulation input.

4.5.5 Front End Conclusion

The front end to PDist shows how a parallel simulator can be utilised as easily as a sequential one. It is a concept prover; the épée environment goes a long way towards providing a more generic and robust input mechanism. The future of process engineering tools relies on them being able to talk engineering to each other. PDist shows some of the benefits of getting them to do this.
The next section now moves on to the back end of PDist. This is by far the most developed and tackles many of the problems associated with managing the kind of information that is likely to come out of large, modular simulators.

4.6 The Back End

This section describes the tools that manage and manipulate the solutions leaving PDist. The section begins by examining the various issues associated with managing large volumes of highly variable data. The remainder of the section de-
scribes each tool and how it tackles the various problems associated with solution management.

4.6.1 Introduction

Dynamic simulators produce vast quantities of solutions at either regular or irregular time stamped intervals. The structure of this information is highly dependent on the problem being solved and the models being used to solve it.

The produced solutions are only useful if they can be presented to the user in a meaningful manner. To do this a solution management and display program is required. This is not trivial to produce. The main problem is not so much related to actual solution display, but to finding and extracting the solutions of interest.

The only way that solutions can be managed is if the overall solution structure can be broken down into definable and hence manageable parts. For a chemical process the first step is relatively simple: split the solutions into groupings based on process topology. With the modular approach being used for this thesis, this process is provided for free. Solution data are already being exclusively managed by the individual simulators being used.

Given that the solutions are now separated into groupings based on process topology, the remaining task is to provide a mechanism for describing and manipulating this more defined information.

Managing Process Unit Solution Information

Earlier text has already shown that in order to simulate a process, an input description is required which defines the process along with its associated static
and model dependent attributes. The structure of the solutions is also highly
dependent on this input information. The number of solutions being produced
is a direct function of process attributes. i.e The number of components and the
number of stages in a column.

A format is thus required which can describe the solutions based on information
taken from the input mechanism and on information about what the particular
models used are capable of producing. As with the input problem, producing a
completely generic process output format is beyond the scope of this thesis. As
with the input mechanism however, PDist has been written with a more targeted
version of what is required.

4.6.2 Simulation Solution Management in PDist

All of the tools which manipulate solutions within PDist obtain information about
the solution structure from a solutions description file. This file is created by the
author of the models being used. For every set of models produced, a suitable
solutions description file must be created. The main purpose of the file is to
declare explicitly what information is being exported from the simulation models
along with recommendations for how these solutions should be managed.

The solution description format is composed of two parts: the solution structure
declarations and the solution management recommendations. Each of these is
now described. An example description file is shown in Appendix C.3 and the
complete syntax is described in Appendix E.1.
The Solution Structure Description

PDist's solutions are packed and exported from the modelling code via the model interface routine pdist_solutions(). Within this function the solutions for a given tray are packed by the models into a single vector. The order that this information appears in the vector is defined by the modeller. The main reason for this is that it caters for the display of any solutions the modeller wishes to export to the outside world.

To allow tools to usefully manage this information, the first part of the solution description file is designed to allow the modeller to declare how many entries are being placed in these solution vectors and to assign informative labels to them. Since the number of entries are usually highly dependent on the problem being solved the description format also allows the use of certain system variables which are automatically instantiated once the problem input file has been read. System variables include: the number of components, the component names, the component molecular weights and densities and the number of distillation plates in the column. Variables are declared by using a $ symbol followed by the variable name enclosed in curly brackets. e.g. In the format the name of component 1 is defined as ${COMP1}.

For every set of models the description file must contain declarations for the solution vectors leaving the reflux, stage and reboiler models. An extract of the format is shown in Figure 4.8.

Each solution entry in the overall vector is called a slot. For each slot in the format a label must be added along with a shortened version which can be used within graphs as keys. The example shown in Figure 4.8 highlights the use of system variables for making the slot numbering a function of the input problem and thus highly flexible. The example also shows how the system variables can
begin stage
    slot 1 for \$\{NCOMP\} "Molefraction \$\{COMP\}" "Liq Mfrac \$\{COMP\}" 
    slot \$(\{NCOMP\}+1) for \$\{NCOMP\} "Molefraction \$\{COMP\}" "Vap Molfrac \$\{COMP\}" 
    slot \$(2*\{NCOMP\}+1) "Total Molar Liquid Flow" "Reflux (Kmols/s)" 
    slot \$(2*\{NCOMP\}+2) "Total Molar Vapour Flow" "Tops Prod. (Kmols/s)" 
    slot \$(2*\{NCOMP\}+3) "Temperature (K)" "Temperature (K)"
end stage

Figure 4.8: Example of Solution Structure Declaration Format

be used to enhance the names given to slot labels.

The structure declaration provides any tool reading the format with information about the amount of data that exists and how it can be located and referenced. The next section of the description format deals with the declaration of how this information can be grouped together and managed in an intuitive manner for the engineer.

The Solution Display Recommendations

The solutions produced by PDist are stored as a series of time stamped solution vectors. Each time slice contains a reflux solution vector, a reboiler solution vector and a solution vector for each stage of the column. Each vector is further described by the solution structure description discussed previously.

The remainder of the solution description file allows recommendations for solution slot grouping to be made under various labels. Different tools can analyse the groupings recommended under the label which applies to their particular function. At present the groupings catered for are purely for the use of display tools. The groupings currently supported are:

3d 3 dimensional plot recommendations.
2d 2 dimensional plot recommendations.

tables Solution tabulation recommendations.

runtime Recommendations for run time graphics to be displayed.

Any tool which reads the solution description format can access the information under any of the groupings shown. They can then either use or ignore the recommendations given. All of the tools within PDist use the recommendations at all times.

Each grouping of recommendations is built out of “contexts”. A context is the label given to a number of recommendations which are associated with a sub part of the overall simulation problem. e.g Solution data which is associated with the context of liquid molefractions. For every context defined, a series of items associated with that context can be declared. Each item must be given a name and a solution vector slot location for where the particular solution data is stored.

Figure 4.9 shows the 3d graph recommendations from the example description file in Appendix C.3. The recommendations shown are grouped into the contexts of flowrates, liquid compositions, vapour compositions and thermodynamics. Within each context the solution management recommendations are listed as items. Each of these items has a number, a label for the item and a description of the solution information to be used. The solution information can be expressed in two ways:

1. As a solution vector slot number
   
   This declares the slot where the particular solution values can be retrieved from.

2. As an algebraic function of slot numbers
In this case the solution value to be used is expressed as an algebraic function of variables, numbers and slot locations. This function is symbolically evaluated as required. The main reason for providing this mechanism is to allow complex solution relationships to be declared. In the example shown, this is used to declare a relationship for converting mole fractions to weight fractions. It is also used as a means of providing temperature plots in centigrade as well as kelvin.

The symbolic maths evaluation facility can be used in other parts of the description file. Any text enclosed with single back quotes will be evaluated and replaced with the result. Again the figure shows an example of its use in slot number evaluation.

The other groups of recommendations are all encoded using a similar format. The main exception is the runtime graphics tool recommendations. The format for this is slightly different due to the simplicity of the current tool being used.

**Solution Format Conclusions**

The solution description format provides a flexible and descriptive mechanism for the PDist model provider to declare and recommend the way in which the tools associated with PDist manipulate the solutions produced. The format is again fairly static and is specifically aimed towards describing the solutions being produced by PDist. The format does however go a long way towards showing that, given time, it should be possible to produce a generic output mechanism that is usable by any dynamic simulator.

The mechanism for describing the solutions leaving PDist has now been described. The rest of this section describes the tools within PDist which read this solution description.
# Calculate the molecular weight sum for use in converting mole fraction
# to mass fraction.

\[
MWTSUM = \frac{\text{MWTSUN} + (SLOT_i \times MWT_i)}{\text{MWTSUM}}
\]

\# Define the 3d Graphing Recommendations

\begin{verbatim}
begin 3d
  context "Flowrates" [:]
    stage menu "Flowrates"
      item 1 "Total Liquid Flowrate" slot (2*$NCOMP+1)
      item 2 "Total Vapour Flowrate" slot (2*$NCOMP+2)
  context "Liq Molefractions" [0.0:1.0]
    stage menu "Liq Molefractions"
      item 1 for $NCOMP "${NCOMP}" slot $INDEX
      item ($NCOMP+1) for $NCOMP "wt frac ${COMP}"
        eval ( ($MWT{BASEINDEX}*SLOT{BASEINDEX}) / MWTSUM )
        "wt frac ${COMP}"
  context "Vap Molefractions" [0.0:1.0]
    stage menu "Vap Molefractions"
      item 1 for $NCOMP "${COMP}" slot ($INDEX+$NCOMP)
  context "Thermodynamics" [:]
    stage menu "Thermodynamics"
      item 1 for $NCOMP "RV of ${COMP}" slot (2*$INDEX+3+$BASEINDEX)
      item ($NCOMP+1) "Temperature (K)" slot (2*$NCOMP+3)
      item ($NCOMP+2) "Temperature (oC)"
        eval ($SLOT'(2*$NCOMP+3)' - 273.0) "Temperature (oC)"
\end{verbatim}

Figure 4.9: Example of Data Management Recommendations for 3D Graphs

### 4.6.3 PDist’s Graphical Tools

The solution output description is used by a number of PDist’s programs:

- The master program
CHAPTER 4. PDIST

- The back end solutions viewer
- The runtime solutions viewer

Each of these programs is described in turn.

The Master Program

The master program is used to set up and collect solutions from the individual simulators which make up PDist. The program uses the solutions description format to calculate the amount of storage required for a particular simulation. The program ignores all of the display recommendations. The master program uses very little of the output description. However, at the end of a given simulation, it does call the back end solution viewer which does use the majority of the recommendations given. This viewer is now described.

The Back End Solutions Viewer

At the end of a simulation, the tool used for general solution display is the back end solution viewer. This viewer can either be executed from a program via a library call or run as a standalone program. For the standalone version a set of stored solutions is required. The viewer can display the solutions either graphically, in 2 or 3 dimensions, or in tables. The graphs and tables available are dictated by the recommendations given in the solution description file. All of the window tools are written in XWindows and all graphics are written in a revised version of TPlot which was originally developed by Eric Fraga [95].

To highlight how the output description file is used by the viewer a number of figures showing screen dumps from the viewer are shown. In particular the
figures show the viewer produced for a three component distillation simulation with model output described by the file given in Appendix C.3.

Figure 4.10: PDist Graphical Display Selection Panel

Figure 4.10 shows the main viewer window through which all display options are selected. The main window contains three row of selectors. The top row contains selectors for 3d graphs, the middle row selectors for 2d graphs and the bottom row a selector for tables. For the graph selectors, there is one for each context defined in the description file. To provide simulation efficiency diagnostics, an extra selector is provided for each graph type to allow timings information produced by PDist to be analysed along with the modelling solution produced.

Figure 4.11: Selection of 3D Context Items

For each row of buttons the mechanism for displaying solutions is slightly differ-
ent. For 3 dimensional graphs, a single graphing window is produced for each item specified in a given context. For this reason each selector on the top row contains a menu of context items which can be selected. Figure 4.11 shows the way the display items are selected by the user. In the menu shown some items have a tick symbol beside them and others a chip symbol. A tick indicates that the solutions being selected are known. A chip indicates that the solutions being selected for display are symbolically evaluated from known solution data.

After a given item has been selected, a 3d graph tool is produced which displays the solution information selected. Each 3d graph displays the required solutions on the vertical axis against time and plate number. The time scale, viewing angle and vertical plot ranges are all changeable. Figure 4.12 shows a 3d plot of temperature through the column.

For the 2 dimensional graphs, a single viewer is produced for each context. For this reason the main window selectors are simply buttons. Once selected a 2d dimensional graph viewer for that context is produced. The items to be plotted are then selected from a menu. All 2d graphs plot the selected item against time. There is an item selection menu for each stage in the column. Every item selected is added to a list. Once the draw button is selected, all of the stored selected items are plotted. Figure 4.13 shows how various items are selected and displayed for the compositions context defined in the output description file.

The final part of the solution viewer is the tabulator. The tabulator is selected using the "Tabled Output" button on the bottom row. Figure 4.14 shows the tabulation window produced. Within the window a complete set of solutions is displayed for a selected time in the simulation. Each row of the table contains solution information relating to either the reflux section, the reboiler section or a particular plate in the column. The columns of the tables display the solutions for all of the items declared in the various contexts of the table display recommendations. The columns contain headers which show which items belong to
The viewer as a whole is extremely usable. It can be used to display any of the solutions leaving the simulation models, provided they have been recommended for display within the output description file. The viewer was found particularly useful by the industrial case study company for viewing changes in the column which had previously been unmeasurable on a real column.

**The Run Time Solutions Viewer**

When PDist is executed with the run time graphics option set on, a copy of all solutions produced is passed from the master program to a run time graphics
program. This program displays the information onto the chosen graphics display.

At present this program can only display 3 dimensional graphs. Again the program only displays graphs for the solutions recommended in the output description file. Figure 4.15 shows the run time solutions viewer in use.

Of all the programs in PDist, this is the least developed. It mainly lacks the ability to display 2 dimensional graphs and tabled output during execution. The addition of these features is relatively simple, just time consuming.
4.6.4 Back End Summary

The back end of PDist has proven to be extremely useful in presenting the usefulness of dynamic simulators to the industrial community. There has been a lot of interest shown in the ability of the simulator to provide complete solution management for the user in a simple and easily used form.
4.7 Summary

PDist was originally written to show that parallelism could be used effectively for dynamic distillation simulation. The current version shows that it is indeed possible to produce parallel simulators which are not only highly parallel but also...
highly usable.

As far as parallelism is concerned, the main work for the future is in connecting simulators like PDist together to produce a single plant simulator. Some work has already been performed in this area. This has resulted in a package called PNet. The next chapter describes the aims of PNet and how it has been used to connect PDist to multiple version of itself.
Chapter 5

PNet: A Parallel/Process Network Simulator

The work on PDist shows that it is possible to utilise parallel processing for the dynamic simulation of distillation columns. It also shows the benefit of adopting a highly modular approach to both model and utility management construction. Given that it is possible to produce similar simulators for other highly complex process units, it should be possible to connect a number of these simulators together to produce a complete plant simulation.

This chapter describes PNet, a Parallel/Process Network simulator. It has been written to demonstrate that this approach is possible. As with PDist, PNet builds upon the parallel simulation theory described in Chapter 3.

5.1 Introduction

PNet was written to demonstrate the use of parallelism for the dynamic simulation of highly complex chemical processes. PNet is not a dynamic simulator in its own right but more of a mechanism for producing one. Figure 5.1 shows what PNet provides.
In the figure there are two shaded boxes. The top box contains the process which is to be simulated dynamically. The process as shown has already been broken down into a series of connected process blocks. The methodology for breaking the process up is based on the availability of specialised dynamic simulators which can be used to simulate the process blocks produced.

Once the process has been broken into blocks, the idea is to load up the local computing hardware with the individual simulators available. Some of these simulator programs may be parallel, PDist being an example. The lower shaded box in the figure shows how the individual simulators would be loaded. Small black boxes represent individual programs, so in the case of the column simulators shown, they are parallel. Once the individual simulators are loaded, the remaining task is to allow them to communicate and act as a single simulator.
PNet is designed to load these simulators and then create and manage the required connections. There are a number of ways in which simulators can be connected. This is now examined, and the eventual mechanism used by PNet described.

5.1.1 Connecting Dynamic Simulators

The theory behind modular simulator construction has already been described in Chapter 3. Briefly, there are three possible ways of connecting dynamic simulators together with a view to utilising parallelism. In general, each simulator is attempting to simulate over a specified time horizon. This horizon represents a global rendezvous point for all simulators. To reach this horizon, each individual simulator may take a varying number of time steps to get there. The three approaches which can be used for multiple simulators to attain this goal are:

1. Explicit Modular Approach
   This is the simplest way of connecting simulators together. Over a given time step each simulator uses fixed inputs. These inputs having been produced by the connected input simulator on the previous time step. Since the input is fixed, each simulator only calculates a given time step once and once only. There is no iteration between simulators. Since simulators can use different time steps to cross a given time horizon, some of the required input variables at a given time may be unavailable. In this case they can be obtained by extrapolating known input values at other times. Matters can be simplified greatly by using a small enough time horizon and forcing connected simulators to only use inputs calculated at the previous time horizon.

2. Implicit Modular Approach
The implicit modular approach is an iterative one. In this case each simulator must use the inputs produced for the current time step by the connected simulators. All of the simulators keep recalculating the time step until there is global convergence over all of the connections. This approach is the same as that used within PDist. It is likely to be more robust, but will consume much more computer time and be more difficult to coordinate. Again each simulator may use a multistep approach across the time horizon and interpolation may be required to provide intermediate stream values.

3. Implicit Lookup Modular Approach

This represents a halfway house between the explicit and implicit approaches. In this case the aim is to avoid the requirement for all of the simulators to iterate together to a globally converged state but yet retain the robustness produced by the iterative method. This is achieved by using historical connection information to guess the connection status for the end of the current time horizon. These guesses are then used as input for the simulators over the time horizon. If the guess was a good one, at the end of the time horizon, each simulator's output should closely match the estimated one. In the event that they are drastically different, the time horizon for future calculations can be reduced. In severe cases this may involve backing up the simulator(s) for the time horizon just calculated. Of all the approaches this is the most difficult to implement, since all simulators must be able to backtrack. However, for processes likely to display discontinuities, this is a useful feature to have.

All of these approaches have their advantages and disadvantages. The aim with PNet has been to attempt to produce a connection mechanism which allows all of these approaches to be used within a consistent framework. With all of the approaches described the simulators remain relatively unaffected. They are always receiving inputs for a given time and either doing a once only calculation, or repeating the same calculation procedure until converged. Either way the
variation between simulation approaches is in the management of connection information.

Given that connection management is the main stumbling block, the question is how should it be implemented? For each connection between simulators there are three possible locations for implementing connection management:

1. At the source simulator
2. At the sink simulator
3. Between the connected simulators

The first two options are unrealistic from a programming viewpoint. In both cases information is being managed by either a source or sink simulator. In order to pass information from one to the other both source and sink simulators would have to synchronise their actions to a large extent. This becomes near impossible to implement when simulators are both source and sinks and also highly connected. Providing flexibility in connection strategies is also problematic. Changing the connection management algorithms would involve the recompilation of any simulator which contains the inbuilt PNet connection management code.

This leaves option 3. Here the solutions are being managed by an intermediate. The only way of providing this, is via an external program dedicated to managing the connection information separate of the two simulators being connected. There are a number of advantages to this approach:

- Simulators can run asynchronously
  This creates less idle time since there are no major simulator synchronisation problems.
Simulator connection/construction is simplified

Simulators simply need to know how to send and receive information from a connection manager. This can be provided via a set of simple routines.

Connection algorithms are detached from the simulator algorithms

The connection managers contain all of the extrapolation/interpolation code. This allows the mathematicians to focus on data management algorithms and the engineers to focus on simulator construction. Again this shows one of the benefits of a highly modular approach.

Changing connection strategies is simplified

Since the connection is managed by a separate program, changing the connection strategy simply involves changing the connection program. There is no recompilation of simulator code required.

Simplified solution management

Since PNet is providing the connections, it is only natural for it to manage the collection of connection information. This is easily provided since all connection programs have to be managed by a central program anyway.

Of the three possibilities, the intermediate connection management approach appears to be the best. It is also by far the simplest to implement and continues the highly modular theme which has been present throughout the majority of this thesis.

The version of PNet produced is designed around intermediate connection managers. The PNet approach to simulator connection is now described.
5.1.2 PNet Approach to Simulator Construction

PNet connects process simulators together via intermediate connection managers. From now on these shall be referred to as "pipes". Figure 5.2 shows how a pipe is used to bind two simulators together.

Figure 5.2: The PNet Connection Manager: The Pipe

The figure shows the management of data transfer across a connection between a process simulator A and a process simulator B. Each process simulator has a number of other connections. These are managed elsewhere by other pipes. In the
Each process simulator program transmits and receives stream connection information via the PNet Interface. For each input/output connection a simulator has, the library allows the simulator to read and write from the particular pipe associated with it. The PNet Interface also provides the simulator with a gateway to a PNet Master program. This is responsible for convergence checking, solution collection and other related management functions.

Once the simulator network has been loaded, every simulator is talking directly to one or more pipe programs. The pipe program itself is built out of three parts. Some of these parts are supplied with PNet. The rest are supplied by the creator of the management algorithm being used by the pipe. The parts are as follows:

1. The pipe kernel

   This is supplied by PNet and is responsible for managing all of the interactions of the pipe with its source and sink simulators. The kernel is essentially a continuous loop program which simply responds to requests for information storage and retrieval. As with the PNet Interface Library the kernel also has a connection to the PNet Master program.

2. The user supplied data handlers

   The user supplied data handlers are responsible for managing storage and retrieval requests from the kernel. The data handling code must be written within a number of kernel callback functions.

3. The data store

   The data store is the area where all current and historical connection data are stored. At present this must be user supplied.

A fuller description of the PNet pipe and how one is produced is given in Section
5.2. For the moment PNet uses a single pipe for every process connection. From an efficiency viewpoint, it may be necessary to manage more than one connection in a single program. This and other related issues are dealt with in Section 5.2.

5.1.3 Other Simulator Connection Issues

The description of PNet so far has focussed on the direct connection of simulators. Although this represents a large part of the overall problem, there are a number of other issues which affect the way in which PNet has been structured. These issues are now described.

Controllers and Control Information

The connections managed by pipes represent the process connections through which actual material is transferred. In a plant this material transfer is managed by process controllers. In reality these controllers act on information read from the process and manipulate a series of control valves. The actual controllers can be situated near the process being controlled or be part of one large centralised control system.

At present PNet has no controller management facilities. Each simulator is responsible for providing the control algorithms it requires for its own process. There is also no mechanism for one simulator to access process information from another. Hence control algorithms cannot span multiple simulators.

Although not yet implemented, control can be added in a similar fashion to material connections. At present a process simulator declares a serious of inlet and outlet connections for pipes. To this could be added a number of outlet analysis ports and a series of control setting input ports. A set of interface
functions could then be written to allow external control programs to read process information from the analysis ports and write control actions to the control setting ports. Unlike with material connections, there is likely to be a wide variety of analysis ports required.

With such a framework it would be possible to build localised control programs for localised controllers and larger control programs which represent the main on site control centre. For all control program produced the binding to the process simulators would be the same. These control programs are more closely related to pipe data managers than the actual simulators. The control programs are essentially data managers in their own right. In this case data is no longer relayed unaltered to a destination port, but is mapped by a control algorithm to another form. For this reason the implementation of a control program would be very similar to that for a pipe.

With the addition of control programs the full structure of PNet is beginning to take shape. PNet is not so much a set of simply connected simulators, but a whole series of specialised programs cooperating together. The structure evolving is beginning to closely represent the structure of an actual plant. The only real difference is that the continuous world is being mapped into a discrete one.

**Solution and Interaction Utilities**

For a dynamic simulator to be usable, the user must be able to interact with it and view the data being produced. The benefits of this have already been shown with PDist.

In PNet, all of the pipes are being managed by a centralised control program: The PNet Master program. It is a simple matter to make this program a simple solution collector for pipe information at regular time intervals. The larger
problem is the collection of solutions from the simulators themselves.

The work on PDist has already highlighted the difficulty associated with providing a generic interaction and solution viewer for process simulators, see Chapter 4. The overall solution relies on being able to fully specify the process structure being simulated, This includes being able to:

- Describe the physical equipment and topology
- Describe the models used to simulate the equipment
- Specify the inputs for a particular set of models
- Specify the output from the models being used

This usually relies greatly on the input specification.

The production of a description format capable of describing all of these features is well beyond the scope of this thesis. As mentioned in Chapter 4, the épée project [94] tackles many of the issues relating to process description and manipulation.

At the moment PNet assumes that individual simulators can provide their own input format and solution display mechanism. For future simulators, this individual simulator management approach is seen as the correct methodology to use. The aim in the end would be to provide a standardised method for providing simulator interaction and solution display. Such a mechanism would have to rely greatly on the overall process description. Again the mechanism would probably be provided through a series of ports attached to simulators, to which specialised interaction and display tools could bind and send and receive information from.
5.1.4 The Current and Proposed Structure of PNet

The majority of the functionality described here is hypothetical. At present only simple pipe managers have been implemented. No control, simulator solution management or interaction mechanisms have been produced. These are still provided internally by the specific simulators being connected. PNet has however been designed with a view to providing these extra features. Figure 5.3 shows the implementation structure of PNet. This includes current and envisaged features. The diagram shows two distinct parts to PNet: The PNet Front End and the PNet Simulator.

The PNet Front End is the mechanism for creating a process and launching a PNet simulation of it. The final product of this creation mechanism is a PNet Simulation Process Object. This contains a complete description of the process, the simulators to be used, the utility programs required, the initial input specification and the solutions which can be produced. There are two possible ways in which this process description can be created: either via a process editing package which has a local data base of available simulators, or via some engineering design system similar to the épée environment mentioned earlier.

At present most of the PNet Front End is fictitious. The only existing part is a simple PNet Simulation Process Object. This specifies the process topology and simulator loading instructions. At present it must be generated by hand. This is fully described in Section 5.4.1.

The PNet Simulator shown in the Figure 5.3 closely resembles the current version. PNet is designed to allow many tools to cooperate. To do this they have to communicate with each other. In the diagram this is labelled as the PNet Communications Layer. This is accessed via a series of specialised interface routines. These routines are specific to the PNet features being used. i.e. talking to
a pipe. The PNet Communications Layer is written in a package called RGC, developed by myself. RGC is designed to be a bridge between PNet applications and a number of various third party communications systems. At present PNet can run on any hardware platform supporting either PVM [93] or CSTools [91].
Similarly any parallel applications written in RGC are executable under PNet. This is described in greater detail in section 5.4.

As shown all of the programs which make up PNet are functionally grouped. At present direct support, via specialised interface routines, is only provided for simulators and pipe programs to intercommunicate. Solution management is provided for pipes by the PNet Master program. Although the other features mentioned above are not yet implemented, the diagram does show how the structure of PNet can be easily extended to support these extra tools. PDist displays the intended functionality of the features not yet supported, especially those related to simulator interaction and solution management.

5.1.5 Summary

The proposed and actual structure of PNet has been described. The current version of PNet aims to demonstrate that it is possible to connect various dynamic simulator programs together via a series of intermediate connection managers, referred to as pipes. It does not yet include support for external control programs, simulator interaction mechanisms or solution management. Supporting these extra features within the structure of PNet is relatively simple. PNet is designed around allowing extra connection types to be added to the simple pipe connections which already exist. The main difficulty with supporting these extra features is in the definition of the data being transferred rather than the actual transferral of it. This is especially the case for solution management and simulator interaction, which require information about the process structure, the models being used to simulate them and what solution data these models produce. Although not supported in PNet, the utility provisions within PDist demonstrate to some degree how a generic set of interaction tools and solution manager programs would fit into a fully working PNet simulation.
The remainder of this chapter now describes the initial version of PNet which has been produced. In particular the implementation of pipes, the creation of connectable simulators and the actual mechanics behind the execution of a PNet simulation are described.

5.2 Creating a PNet Pipe

The pipe is the main building block of a PNet simulation. Without pipes it is impossible for one simulator to communicate with another. This section describes the functionality a PNet pipe aims to provide, and how pipes are implemented and created within the current version of PNet.

5.2.1 The PNet Pipe

Each pipe is responsible for managing a single simulator to simulator connection. It is quite likely that, for efficiency reasons, each pipe program should be capable of handling multiple connections. For the moment this has been ignored since once PNet is fully working with single connection pipes, the effort required to produce multi-connection pipe programs is relatively trivial.

Before describing the implementation of pipes, it is first important to understand exactly what each pipe is supposed to provide in terms of functionality. The current version of PNet requires the following:

1. Full Simulator Connection Data Management

   The main function of a pipe is to manage connection data. This involves storing the time stamped data being produced by the source simulator and retrieving the time stamped data being requested by the sink simulator. Since each pipe is a separate program, connected simulators can run asyn-
chronously and thus the data storage time intervals do not necessarily have to correspond to the requested ones. The local data management algorithms should be able to provide the requested data by extrapolating or interpolating on the data which has actually been stored. This process is simplified somewhat by the requirement for all pipes to work over a commonly recognised time horizon.

2. A Global Convergence Mechanism

Some of the parallel connection strategies require there to be a global convergence mechanism. As described in Chapter 3, all of the connections are essentially tear streams. For connection strategies requiring iteration between simulators, each simulator must be able to determine when the whole process network of simulators has converged. To do this each pipe must be able to register its convergence status to a centralised manager, which can then decide when the whole network has converged. Once this occurs each simulator and pipe must then be informed that a globally converged state has been reached.

3. Information Flow Management

Information flow around the simulator network is totally controlled by the pipes. There is no global pipe controller in PNet. Each pipe program must thus be able to control the way in which connected simulators can store and access information. This not only allows connections to be managed more efficiently, but allows the PNet to behave in specialised ways. One example being that it is a simple task to make the PNet act as a sequential modular flowsheeting package.

4. PNet Shutdown

Global shutdown of PNet is a cooperative event. Neither pipes or simulators are designated as having overall control. This allows all simulators and pipes to keep running until a general agreement on shutdown is reached.
This approach also allows simulators which have finished to shutdown while others are still running, thus freeing up hardware resources immediately for other users. To provide this feature, pipes must be able to contribute to shutdown decisions and also be able to work out when connected simulators are finished.

For the moment these features are all that is required. For the future the main extensions would cater for:

- **Illegal Storage Detection**

  At present, simulators are free to send any data they wish to a pipe. There is no mechanism for a pipe to reject incoming information. This presents a problem when the down stream simulator wishes to put constraints on particular input data slots. In particular this is likely to be with respect to flowrates. Managing such restrictions would involve the sink simulator having to register its input limitations to the source pipe. The pipe could then reject storage requests that were unacceptable. A simulator whose storage request was denied, would have to cope with the change in its output from that calculated to that allowed.

- **Remote data access**

  Allowing remote access to connection information from control programs and other simulators not actually connected to the pipe. As mentioned before, this is relatively simple to implement within the current structure.

Depending on the simulator connection strategy being used and the type of simulation being run, different pipes will be required. It would be possible to build a single pipe to cater for all connection strategies, but this would be inefficient. This requirement for many pipes means that their construction must be simple. The framework for pipe construction must be relatively standard, yet allow the
above mentioned pipe features to easily built in a flexible and intuitive manner. How this has been achieved is now described.

5.2.2 Pipe Creation and the Pipe Interface Routines

All pipes must be built using a set of pipe interface routines. These routines are designed to hide the underlying communications code from the pipe writer leaving them free to concentrate on connection management. At present these are only available for the C language.

Figure 5.4 shows the structure of the pipe interface routines. The interface routines are grouped under the functionality they provide. Each box contains a group of interface routines. The arrows indicate the execution path usually taken from one set to another.

The majority of the routines are part of the PNet library and can thus be called within a program. Some however must be supplied by the pipe writer. These routines are called by the pipe kernel during execution and through these the pipe writer must implement the particular connection management strategy for the pipe.

The main programming code for each pipe must be supplied by the pipe writer. From this code the various pipe interface routines can be called. The routines are grouped as follows:

- Initialisation/Shutdown Routines

  These routines are the first and last routines to be called in a pipe main program.

  - pipe.init

    Initialises the pipes communications layer and underlying structure
Figure 5.4: Structure of the Pipe Interface Routines

- **pipe_exit**
  Tidies up the communication layer ensuring clean pipe program shutdown.

- **Pipe Information Routines**
  In order to efficiently implement the various connection strategies, each pipe needs to know various things about itself. These routines provide all the information available.

  - **pipe_isa_feed**
    Returns TRUE if the pipe is a feed. i.e Has no source simulator.

  - **pipe_isa_recycle**
    Returns TRUE if the pipe is a recycle.

  - **pipe_isa_product**
    Returns TRUE if the pipe is a product. i.e Has no sink simulator.

  - **pipe_datasize**
    Returns the expected size of the stream data being managed.
- pipe_number
  Returns the unique number given to the pipe by PNet.

- pipe_name
  Returns the name of the pipe specified in the PNet Process Input Object.

- pipe_finish_time
  Returns the duration of the current simulation.

- pipe_histstep
  Returns the interval over which solutions are collected.

- pipe_timestep
  Returns the current time horizon for the simulation.

- pipe_import
  This is a specialised function which can be used for importing special simulation parameters. Imports are described later in Section 5.4.1.

• The Pipe Kernel Routine

  pipe_main_loop is the routine which implements the pipe kernel. This routine never returns until the pipe has finished.

• Pipe Kernel Callback Routines

  The main features of the pipe are implemented via a set of callback routines. Each routine must be supplied by the pipe writer. The callback routines are called by pipe_main_loop at the appropriate times.

  - pipe_solutions
    This is called when solutions for a given time are required. The code in the routine must retrieve the data for the time requested. Solutions are never requested for a time greater than any that have yet been stored.
- pipe_store
  This is called when the kernel receives a request for storage from the source simulator.

- pipe_request
  This is called when the kernel has received a request for information at a given time from the sink simulator. The code in this routine must retrieve the information from the local store or get it via extrapolation/interpolation. If the request cannot be satisfied, then by returning FALSE from this routine, the kernel will queue the request and recall this routine once new data has been stored by pipe_store.

- pipe_converged
  The pipe kernel contains code for registering and deregistering its convergence state. Once all of the pipes in the simulator network converge, the pipe kernel calls this routine. In this routine all of the code relating to convergence management is usually placed.

• Pipe Kernel Control Routines
  This final set of routines are used to control and manipulate the way in which the pipe kernel behaves. These routines are usually called from the kernel callback routines. They can also be called from the main program for initialising the pipe kernel before its started with pipe_main_loop.

  - pipe_stop_store
    This stops the kernel from accepting store requests. It forces any source simulator to wait.

  - pipe_start_store
    This restarts the kernel receiving storage requests.

  - pipe_stop_request
    This stops the kernel receiving requests for data from a sink simulator. It forces the sink simulator to wait.
- pipe_start_request
  This restarts the kernel receiving data requests.

- pipe_set_converged
  This tells the kernel that the pipe has converged. The kernel then tells
  the PNet Master program. This monitors all of the pipes convergence
  states. This routine only needs to be called once per convergence.

- pipe_unset_converged
  This tells the kernel that you are now unconverged. If this is called
  before the pipe receives a global convergence from the PNet Master,
  the kernel informs the PNet Master of the change. The routine is only
  to be used for unsetting a previously set convergence.

- pipe_finished
  Any PNet simulation can either be shutdown by the pipes, the sim-
  ulators or a combination of both. This routine indicates to the pipe
  kernel that the pipe has finished as far as the programmer is con-
  cerned. However, the pipe cannot simply shutdown. It must wait
  until the neighbouring simulators are either informed, or have com-
  pleted themselves. On finishing the pipe_main_loop routine returns to
  the calling program. The mechanism also helps in propagating pipe er-
  rors to connected simulators, thus enabling clean and efficient program
  shutdown.

The routines described allow pipes to be easily constructed, with the majority
of user code having to be supplied within a few routines. The pipes which have
been constructed for this thesis are now described.
5.2.3 Example Pipes Created

Three example pipes have been written. The following description is designed to point out the purpose of the pipes which have been developed. The results from their actual use in PNet simulations are described in Chapter 6.

Pipes have been produced to allow the creation of the following simulators:

- A Sequential Modular steady state simulator
- A parallel dynamic process simulator
- A hybrid steady state/parallel dynamic simulator

Each of these is now described.

A Sequential Modular Steady State Simulator

The main use for dynamic simulators is for analysing process changes around the steady state operating point. If the steady state models are available for the process being simulated, and are usable, it is a lot faster to use them to estimate the steady state point than to use dynamic models.

The steady state pipe is designed to connect a series of steady state simulators together. The connection strategy used in the pipe, forces the overall process network to act like a sequential modular flowsheeting package. The sequential approach is the only alternative available since PNet cannot be used to produce an equation based flowsheeting package. It can however connect equation based flowsheeting packages together in a sequential modular fashion. The aim here is to produce something useful. It is unlikely that any benefit from parallelisation will be obtained. The reasons for this have been described in Chapter 3.
Another reason for developing this pipe is its usefulness in testing PNet. Implementing the steady state methodology requires the use of almost all of the features that the pipe interface routines can provide. In particular the data flow control routines, global convergence routines and global shutdown routines are extensively used. The resulting test simulations are also an excellent test of the process communications since the sequential modular method relies on data flowing efficiently around the whole process.

Sequential modular flowsheeting requires solving the steady state models of the various processes being connected in a strict order. This order is usually defined as the sequence of units obtained by starting at the front of the process and moving downstream. Certain connections are regarded as tear connections: these are usually the recycles of the process. The sequence of solution is repeated until these tear connections have converged. Rather than direct substitution, convergence algorithms are quite often used on the tear connection data. The pipes produced here can find out if they are tear pipes and as such can easily implement such algorithms if required.

This sequential calculation sequence can most easily be obtained by making each pipe only allow requests for data from the sink simulator once new data has been received from the source simulator. This being implemented using the various pipe data flow control routines. At the start of operation, each pipe stops the sink simulator from requesting data. The only exceptions are feed and recycle pipes. Feed pipes have preset data and always allow requests since that is their function. Recycles usually represent the tear streams in a process. They also represent a back flow of information. In order to start the desired flow of information within PNet, the recycle pipe data must be estimated and requests allowed within the pipe. From this point each pipe is only allowed to either store data or receive requests for data. This condition is alternated as data is stored and received. This is implemented, as would be expected, in the pipe_store and pipe_request
callback routines executed by the pipe kernel. The upshot of all this is that only one simulator is executing at any one time, except where branches in the process allow for parallel execution.

Each pipe continues to operate until it has locally converged. This is registered using the routine pipe_set_converged with the PNet master process. The pipes then keep going until the callback routine pipe_converged is called by the kernel, indicating that the complete set of pipes have converged. Each pipe then registers its intent to quit with pipe_finished.

The actual simulators used with this pipe must keep going until the pipe network has shutdown or until they pick up the new convergence state. The former is picked up using a PNet library call which is described in the next section. There is no real need for the simulators to use both methods of pipe completion detection; using both simply results in slightly faster shutdown.

Overall this pipe works well, as will be shown in Chapter 6. This pipe has been used as the main testbed pipe for making sure that PNet is connecting and passing information around correctly.

A parallel dynamic process simulator

This next pipe is the first pipe to be produced for building a parallel dynamic simulator with PNet. It has been designed to implement the explicit modular parallel connection strategy highlighted in section 5.1.1. As yet no iterative implicit modular or implicit lookup modular dynamic pipes have been created. However, this pipe has proved extremely good at connecting the type of simulators so far used. The other pipes are likely to be required where the dynamics of the connections are faster than those tested.
This pipe is a lot simpler than the steady state one. Gone are the complexities of convergence and information flow control. It also has no complex interpolation/extrapolation algorithms at present. Any connected simulators are forced to either do this themselves or use a globally recognised time step.

The main function of the pipe is to simply store and relay data from one simulator to the other. Each pipe must always be ready to store data and to give it out if available. If data is unavailable, the sink simulator must wait.

The connected simulators for a given time step from time $t$ to time $t + \delta t$ must always use the input conditions calculated for time $t$. To allow all of the simulators to effectively work in parallel, it is important that each simulator registers its outlets for time $t = 0$ immediately on startup, thus making sure that all initial requests for data can be satisfied.

Shutdown, in this case, can either be detected as before by monitoring the status of the pipe network, or simply keeping a local count of time and finishing at the appropriate moment.

Although simple, this pipe works extremely well. In particular it is well suited to connecting simulators across dynamically slow connections. Results of this pipe's use for connecting multiple PDist simulators together are given in Chapter 6.

A hybrid steady state/parallel dynamic simulator

This final pipe is a hybrid of the last two. Instead of the steady state pipe simply exiting on completion, it reverts to having the functionality of the explicit modular pipe. Again, the use of this pipe with PDist is described in Chapter 6.
5.2.4 Summary

The mechanism for creating a pipe has been described. It is possible to build highly complex intermediate data managers within the current framework. Three test pipes have been created and described. Although simple, they have already been used to build some complex simulators from various smaller ones. These are fully described in Chapter 6.

5.3 Creating a PNet Connectable Simulator

Simulators running under PNet, communicate to the surrounding pipes via the PNet Interface routines. Unlike pipes, simulators do not require a large amount of system code to operate. This has been made possible by moving the majority of the system management code into the pipe kernels. All that is required is a series of routines which allow each simulator to communicate and use the facilities each pipe kernel provides.

5.3.1 The PNet Interface Routines

The PNet Interface has to allow PNet simulators to do the following:

- Request inlet information
  
  Involves asking the connected inlet pipes for stored information at a particular time.

- Store outlet information
  
  Involves asking the connected outlet pipes to store information at a particular time.
• Access the global convergence manager.

Iterative and steady state simulators require information about the status of the complete network. This information can be easily obtained from the connected pipes since each pipe knows the status of the complete network.

• Access the network status and control shutdown

As mentioned before, PNet shutdown can be initiated by the simulators or the pipes. Either way there must be routines supporting both cases.

• Allow parallel processing for the simulators

Each pipe can only be accessed by a single program at each end. For simulators designed to run in parallel, the simulators input and output connections are likely to be distributed over a number of programs. The interface routines must cater for this.

• Allow simulators to access information from the input description

To simplify matters, connections are referenced via numbers. i.e If a simulator has 3 inlets, they are referenced as inlets 1,2 and 3. At present the only mechanism for a simulator to distinguish between connections is from the order. Ideally there should be some mechanism for determining what exactly each connection is with respect to the process being simulated. By using connection numbers, it is a simple matter to provide support for the majority of programming languages available. At present both FORTRAN and C are supported. As with the Pipe Interface routines, the PNet routines are grouped under the functionality they provide. The current set of PNet Interface routines are as follows:

• Initialisation/Shutdown Routines

These routines are the first and last routines to be called by a PNet simulator.
- `pnet_init`

  Initialises the local communications layer. The routine assigns a number of information parameters. These include the number of inlet pipes, the number of outlet pipes, the expected data size being transmitted via the pipes, the various time sizes and the name assigned to the simulator.

- `pnet_exit`

  Cleanly shuts down the communications layer and pipe connections.

- PNet Information Routines

  These allow the simulator to obtain information about itself and from the input description of the process. At present the local information about the simulator is returned in the `pnet_init` function. As for accessing information from the input description, this is achieved via the routine `pnet_import`. It is analogous to the routine `pipe_import` in the Pipe Interface. The exact function of this routine is described in Section 5.4.1.

- Pipe Connection Initiation

  These routines open the gateways to the simulators connected pipe kernels.

    - `pnet_open_in`

      Opens a connection to the requested inlet pipe

    - `pnet_open_out`

      Opens a connection to the requested outlet pipe

    - `pnet_open_all`

      Opens connections to all connected pipes. This is the most common routine used for simulators which are not parallelised.

- Pipe Kernel Communications Routines

  These routines manage all of the kernel requests from the simulator. The routines are:
- **pnet_rx**
  Receive routine. Used to request time stamped information from an inlet pipe.

- **pnet_tx**
  Transmission routine. Used to send time stamped information to an outlet pipe.

- **pnet_convergence**
  Returns the current convergence level of the simulator. Obtains this from the connected pipes.

- **pnet_isup**
  Returns TRUE if the connected pipes are up and running. This routine is usually used to monitor for pipe initiated termination.

- **pnet_shutdown**
  This routine is used to inform all connected pipes that the simulator is shutting down. It does not return until all of the pipes have been informed. After this call the kernel routines are unavailable.

Figure 5.5 shows how a simple dynamic mixer simulator could be constructed using these routines. In the example the actual simulation step is being carried out by a routine called dynamic_mixer(). The example shown is in Fortran.

The current set of routines described are totally dependent on the pipes for information. The communication link to the PNet Master program is unused. This is being reserved for implementing the time management functions required to allow simulators to back up and change time horizons etc. Once implemented the routines above will be extended appropriately.

Although simple, the routines effectively allow simulators to bind together and use the pipes effectively. The routines have been fully tested using the pipes
include 'PNet.inc'

*** Define maximum sizes and variables
integer MAXINS, MAXDATASIZE, nins, nouts, streamsize
parameter (MAXINS=10, MAXSTREAMSIZE=30)
double precision inlets(MAXSTREAMSIZE, MAXINS), outlet(MAXSTREAMSIZE)
double precision endtime, histdt, tm
character*100 name
external dynamic_mixer

*** Initialise PNet ***
call pnet_init(nins, nouts, streamsize, endtime, histdt, name)
if (nouts.ne.1) then
    print *, 'mixer error: only 1 outlet allowed'
call pnet_exit()
    stop
end if

*** open up all the connections to the surrounding pipes ***
call pnet_open_all()

*** Go into main mixer calculation loop ***
tm=0.0d+0
10 continue

*** get the inlet streams ***
do 20 i=1, nins, 1
    call pnet_rx(i, tm, inlets(1,i), streamsize))
20 continue

*** Perform the dynamic mixing process ***
call dynamic_mixer(inlets, nins, outlet, streamsize)

*** Move on to the next time step ***
tm=tm+histdt

*** Send out the mixed stream for new time ***
call pnet_tx(1, tm, outlet, streamsize)

*** Am I finished ***
if ((tm+histdt/2.0).lt.endtime) then
    goto 10
else
    *** Clean up PNet ***
call pnet_shutdown()
call pnet_exit()
endif
end

Figure 5.5: Simple Dynamic Mixer Using PNet
described in the last section. In particular the routines have been used to convert
an existing flowsheeting package to be PNet compliant and to connect multiple
PDist simulators together. The latter is an effective demonstration of the use of
the PNet routines to allow a parallel simulator to become part of an even bigger
one.

5.4 Describing and Executing a PNet Simulation

The difficulties associated with describing the input and output of a process
simulation have already been discussed. The ideal mechanism for PNet has also
been described in Section 5.1.4. This section describes the input mechanism which
is currently being used for PNet, and how it supports the connection strategies
already described and the execution of simulators on various types of hardware.

5.4.1 Describing a PNet Process

The PNet loader and PNet Master program find out about the simulation being
run from a PNet input description file. This file completely describes the topology
of the process network being simulated, the simulator programs which are being
connected and the stream data which is to be transmitted. All simulator input,
output and interaction must be managed by the individual simulators. Example
input descriptions are given in Appendix D and the exact syntax is described in
Appendix E.3.

The input file is built out of sections. Each of these sections describes a different
part of the simulation being run.
The Data Exchange Description

All of the connected programs in PNet need to know the amount of data that they are supposed to be passing between each other. Likewise the solution manager needs to know what each data slot being passed represents, so that it can effectively display the solutions produced.

This information is declared in input description as a series of number name pairs. The number refers to the slot index and the name is the name to be used in displaying the final solutions.

The Process Network

Each process network is described by the processes connected together. For each process the input pipes, output pipes and simulation program to be used are declared along with any input parameters required. Figure 5.6 shows a PNet process declaration. The pipe declarations are very simple. Only the names of the pipes are specified. These names must be unique. A process connection is defined where a process outpipe has the same name as an other processes inpipe. A useful extension would be to add a position name which helps distinguish the various pipe connections from each other. e.g. A column could label its outpipes as having position names “Tops” and “Bottoms”. However, where these pipes meet another simulator as inpipes they may have the position name “Feed”. For complex simulators with complex connections this feature would be extremely useful to the programmer allowing them to easily distinguish which pipes are which.

The program definition is designed to allow the declaration of a parallel program. All of the programs to be executed are registered as processes: the computer kind. Each process is given the program name, the arguments it takes and the
# A Distillation Process Description

begin process "Column 101-1"
    inpipes ["1"]
    outpipes ["2","3"]

    program "DistSim"
    process 0
        exec "column"
        args ""
        imports ["int NumStages","40",
            "int NumFeeds","1",
            "int NumComps","3"]
        proc_type "sun4"
    end process

Figure 5.6: Example PNet Process Description

processor type. The PNet loader works out where best, out of the computing resources available, to place the various programs declared at runtime.

Every program description can also contain a series of so called “imports”. These are declarations of the variables you want exported to the program at runtime. On program execution the data can be imported using the routines pipe_import and pnet_import for pipes and simulators respectively. As yet there is no mechanism for declaring pipe imports. Only the system defined pipe imports can currently be accessed. Each import has two parts: the import type/name and the actual data in string form. The import types supported are integers, reals, double precisions, vectors of these types and strings. The mechanism was originally designed to simplify the initialisation of loaded programs in RGC, but was later extended for use in PNet.
The Feed Data Declarations

The final part of the description file is used to declare data which feed pipes send to their sink simulators at every data request. The format is simply the name of the pipe followed by a list of real numbers. The list must contain the same number of entries as the data exchange description.

5.4.2 Supporting Parallel Simulators

The PNet description file is designed to allow parallel programs to be declared. The only specific hardware declarations required in the description file is the processor type that each simulator program should id run on.

As already mentioned, PNet is written in RGC (Rotations). All programs must be compiled with the communications library RGC provides. RGC is designed to support various hardware specific communications layers provided for the hardware in question. Unfortunately this means that for any one program, a separate compiled version is required for each hardware type and communication hardware.

To make program location simpler, PNet insists that programs produced are given specific name extensions. In the description file, each program is referred to by its basename. The actual program names produced must be extended to include the communications software name and the processor type it is compiled for. i.e. For a SUN4 workstation, a program mixer would be compiled as mixer_PVM.sun4 for PVM and mixer_MKCS1.sun for CSTools. This same naming convention applies to pipes as well as simulators. The actual hardware programs are run on is determined by PNet at runtime and allows it to load up the local hardware as efficiently as possible.
The local hardware available is declared in RGC databases. There is a database for each supported communications layer. Each database contains all of the supported hardware types, how many of each type are available, what their names are and if they are remotely usable through the network.

The current loader attempts to use all of the hardware available by default. The overall usage can however be limited at the command line. The actual execution details of PNet are now described.

5.4.3 Execution of PNet

Until now, all of the text has focused on describing how PNet programs are created and declared. Within the definition of these programs, reference has been made to a number of routines which can extract knowledge about each program's setup and place in the overall process network. This section describes how PNet is actually executed and how this information is picked up by the various programs launched.

Creating a PNet Input Description

Before executing PNet, an input description file is required. At present this must be created by hand. Writing the general process descriptions is relatively simple. Unfortunately the simulator program definitions can be extremely involved, especially for parallel programs. To work around this problem, the input description has been extended to allow parts of the description to be filled in by external builder programs at runtime. Figure 5.7 shows a modified description file.

Figure 5.7 shows the program description replaced by a system call description. When PNet is executed, a preprocessor is run over the input description sup-
# A Distillation Process Description

begin process "Column 101-1"
  inpipes ["1"]
  outpipes ["2","3"]

# system call to create a 9 processor PDist simulator program
# for use with CSTools

  system "pdist -p 9 -config MKCS1a DEFAULT -d -map PNET setup.pdist"

end process

Figure 5.7: Example Input Description with System Call

plied by the user. Wherever a system call is found, the text is removed and replaced with the text produced when the system call is executed. In the figure shown a PDist command is being used to create a nine processor PDist program description.

This feature has proven very useful for setting up simulations. Manipulating the system calls is much simpler than manipulating the 20+ lines of description text required to setup simulators like PDist. To highlight the saving, an example loader description created by PDist is shown in Appendix D.2.3.

Running the Simulator

Once an input description file has been created, PNet can be executed. The command used to run PNet is extremely simple. An example command is shown in Figure 5.8.

The loader command expects a number of arguments. These specify the limits on processor usage, the pipe program to be used, the finish time of the simulation, the history time step and the input description file to be used. Example declarations
command: pnet -p sun4 3 -pipe std-dyn -f 1000.0 -d 5.0 setup.pnet

The arguments are as follows:

- `p sun4 3` : Limits the usage of SUN4 workstations to 3.
- `pipe std-dyn` : Sets the pipe program to use to be std-dyn
- `-f 1000.0` : Sets the finish time to 1000.0 seconds.
- `-d 5.0` : Sets the time step to 5.0 seconds.
- `setup.pnet` : The name of the input description file.

Figure 5.8: Example PNet Loader Command

of all of these are shown in the figure.

Once executed the PNet loader performs the following:

1. Parses the input description file
2. Builds a network graph from the list of processes created
3. Builds up a list of the programs required
4. Checks that all the programs required exist
5. Analyses the process network graph, locating recycles etc.
6. Assigns the requested pipes to the network connections
7. Builds a hardware load map for simulators, pipes and utilities
8. Builds an RGC loader description

All of the information about recycles, names etc. are encoded for each program using the import mechanism previously discussed.

9. Loads and starts up the network of programs using RGC
10. Exits when all programs loaded are finished
Once loaded with RGC, the simulator is up and running. At the end of a given simulation, the only remaining program is the PNet Master program. This displays all of the solutions collected. Once this exits, the PNet simulation is complete.

### 5.4.4 Solution Display

The main solution collector is the PNet Master program. All solutions are currently shown as time stamped tables of numbers. An example of the output is shown Figure 5.9. It would be nice to be able to view these results graphically in a similar manner to PDist. However viewer allows all solution output to be viewed and is suitable for making sure that the initial simulator is working.

![Figure 5.9: Screen Dump of the PNet Solutions Viewer](image-url)
5.5 Summary

PNet is designed to prove that a number of simulators can be connected together to form a larger one. An initial test version has been developed which uses intermediate connection managers, or pipes, to do this. These pipes are created using a series of specialised routines. Likewise the simulators being connect communicate with the pipes via a similar set of routines.

The interface routines have been designed to allow both pipes and simulators to be easily constructed. Using these a number of test pipes and simulators have been built. The next chapter describes and examines the various tests which have been carried out using these pipes and simulators within PNet. The chapter also describes all of the test results obtained using PDist.
Chapter 6

Evaluation

This chapter examines the performance of the dynamic simulators PDist and PNet. Since the simulators are conceptually different, each is dealt with in turn. For PDist the main interest is in showing the usefulness, robustness and efficiency of the simulator. For PNet the main interest is to prove that the connection strategy works, is usable for complex simulations and that the parallelism can provide the performance required for the future.

6.1 PDist Results

There are two questions to be asked about a dynamic simulator: what can it do and how well can it do it? The first obviously takes precedence over the second. A large amount of research has been carried out elsewhere on parallelism using very simple problems. There is often no allowance for how the algorithm is extendible to more realistic problems. With PDist, however, this is not the case. All of the work carried out has been towards producing a simulator which is not only usable but capable of tackling real problems.

This section begins by examining the usefulness, robustness and usability of
CHAPTER 6. EVALUATION

PDist. This is followed by an examination of the actual performance of the simulator on parallel hardware. Finally, the overall conclusions from the tests are summarised.

6.1.1 The Usefulness of PDist

PDist has been tested on a number of problems using a number of models. The actual models have already been described. Recapping, the models produced are as follows:

- Conventional Distillation Models
  These are a set of models produced by myself to simulate conventional distillation columns. Conventional meaning that they contain a reflux drum, tray column and reboiler. The models contain simple hydraulics, full v.l.e via the PPDS [88] physical properties package and use an implicit, and hence iterative, integration algorithm.

- Conventional Distillation Models for Startup
  These models were written by Vladimir Vasek [85]. They are designed to allow startup conditions to be simulated. The models use an explicit integration algorithm and contain complex tray hydraulic models. The original simulation program produced by Vladimir was dismantled and restructured to fit into the PDist Interface structure described in Section 4.4.2. Some of the results using these models are described in [72].

- Industrial Case Study Models
  These models were developed to simulate a reactive azeotropic distillation column. The problem was tackled because the company concerned could not get any commercial packages to solve the problem. The models used were an extension of the conventional models already described. To these
were added a set of liquid/liquid separator models and a reactor/reboiler model. A particular feature of the models is that they incorporate a wide variety of data sources. The company provided NRTL coefficients for the liquid/liquid separator and full kinetic information for the reactor. These were used alongside PPDS, which was again used for the tray v.l.e calculations.

All of the models described are focused at particular problems. However the models do cover a wide range of possible columns and control arrangements. The models can also simulate highly non-ideal separations. The industrial case study column represents about as non-ideal a problem as you can get. From a PDist testing viewpoint, the models also use the majority of the features PDist provides. In particular the models use both explicit and implicit integration strategies. For the industrial case study a mixture of the two is in fact used.

Overall, PDist is more than just a simple exercise in exploring parallelism. The package can be easily tailored to simulate highly non-ideal and complex distillation columns. The successful use of PDist for the industrial case study highlights its usefulness in this area.

6.1.2 The Robustness of PDist

With PDist, the overall simulation is as robust as the individual simulation blocks. If any one of these blocks does not converge under certain conditions, then the whole simulation will not converge. Similarly, any block not designed to handle possible discontinuities is likely to cause convergence problems elsewhere. e.g. Two phases appearing and disappearing in a liquid/liquid separator can cause oscillation in an iterative solution strategy.

Making a simulator robust relies on making the models robust. For equation
based simulators, this has always been a problem since all of the equations are being solved simultaneously and managing discontinuities is difficult. The advantage of the modular approach has always been that by subdividing the overall problem, the robustness problem is also subdivided. Each sub block is a single process whose outputs must be calculated from its inputs. The solution method used to perform this task is completely optional and allows specific problems to be solved with targeted solution methods. Since PDist is a modular simulator, the robustness advantages are inherent. Any discontinuities can be managed locally where they occur before they affect the rest of the system.

This theory bears out in practice. It has proven very difficult to make PDist fall over without setting it up to simulate problems with unrealistic inputs. In the cases where it has fallen over, it has been for expected reasons: e.g. too high a time step for the explicit models, discontinuities not yet handled by the models and errors in external physical properties packages.

The biggest success has been with the industrial case study models. The task here was to find the steady state for the column and analyse the effect of various input perturbations around this steady state point. The main interest was in finding out how the entrainer distribution and reaction was affected. For various reasons relating to the operational mode of the column, the steady state point had to be dynamically simulated. Until the problem was tackled with PDist, this steady state point had not been successfully found. A number of commercial and inhouse packages had apparently been used. The models for PDist took about 2 months to write and test. The steady state point was successfully located and a number of requested test cases run. The results produced were reputedly very close to those expected. Although a highly non-ideal column, the simulations proved extremely robust. Simulations were run for over 24 hours on some of the test cases. PDist was eventually purchased by the company.
6.1.3 The Usability of PDist

PDist has been written to show that parallel hardware can be used for dynamic simulation in as usable a fashion as sequential hardware. A lot of effort has been put into PDist to demonstrate this. This is especially the case as far as the input programming, interaction and solution display is concerned. The flexibility of input is best described with reference to an example input file. Appendix C.2 shows how one of the industrial test cases was set up. In the example, the programming section of the input file is being used to set up a number of changes in the column’s inputs. The modelling code was designed to allow the feeds to be changed either as oscillations or step changes around a base setting. The example shows how the parameters to this code have been preprogrammed to produce the desired effect.

Overall the input mechanism has proven extremely effective. There is still a lot of work required in the general area of simulator input. This is however a thesis project in its own right. Hopefully the input for PDist highlights the potential of moving towards standardising on the description of chemical processes. Again this is one of the aims of the épée project [94].

The interaction mechanism has also proved useful. All simulators need to be controllable. If the changes required are not known in advance, then the ability to interact with the simulation is essential. This is of particular benefit when simulators are being used to investigate control solutions to particular process changes or operators are being trained. The interaction mechanism produced for PDist works well but could be improved. To implement an advanced interaction mechanism you need an advanced process description to allow complex changes to be made and fully described. Although simple, the current mechanism displays all of the qualities required: all variables in the setup can be easily changed and recorded using the standard input description. The method of implementation
is correct. The main flaw is that the data structure being manipulated is not as complex and flexible as that really required.

The final utility within PDist is the runtime and post runtime solution viewing. Of all the utilities, this is the most highly developed. It is also the utility which has generated the greatest interest in demonstrations. The ability to be able to monitor what a simulator is doing in real time, is seen as essential by the industrial community. The ease of display of solutions at the end is also seen as highly desirable. The feature most appreciated has been the ability of the solution viewer to automatically tailor itself to display the solutions produced by a given set of models in a predefined and context sensitive manner.

Generally speaking, PDist is extremely usable. It displays many features not yet available in most commercial simulators, although things are slowly beginning to change. If anything, the features within PDist should highlight to industry what modern computing can offer them and what they should be asking for from the commercial packages they are currently so reliant upon.

6.1.4 Gain from Parallelisation

Given that PDist is usable for real problems, does the parallelisation produce the reduction in execution times expected? The remainder of this section describes and analyses the results produced from running PDist on various numbers of processors.

Defining Efficiency

The performance of a parallel program is usually defined by either speedup or efficiency. The speedup of a parallel program running on \( p \) processors is defined
by:

\[ Speedup \ S_p = \frac{Execution \ time \ on \ single \ processor}{Execution \ time \ on \ p \ processors} = \frac{T_1}{T_p} \quad (6.1) \]

The efficiency is defined by:

\[ Efficiency \ E_p = \frac{S_p}{p} = \frac{T_1}{pT_p} \quad (6.2) \]

The execution time on a single processor can be taken as either the execution time of the best sequential algorithm or the execution time of the parallel program on a single processor. Comparison with the best sequential algorithm is preferred, but this is not always available.

In the case of PDist, the best sequential algorithm is not available. The models being used are designed around the input output model of calculation to allow parallel execution. The most efficient sequential modular algorithm does not use this input/output model but reverses it somewhat. The two algorithms are only equivalent when explicit integration is being used on a per tray basis. The full theory behind this has already been discussed in Section 3.2.

For the results presented here, the parallel runs are compared with those run on a single processor. This is allowable since the modelling methodology used has proven to be extremely useful and robust. Any improvement in execution of this approach through parallelism still represents a distinct benefit.

There is another problem relating to the analysis. At the moment the reflux and reboiler sections are simulated separately from the stage sections. The reboiler and reflux programs normally use the same processing time as a single tray stage program. This means that in a parallel simulation, the stage simulations use much
more processing power than the reflux and reboiler simulations. To combat this the reflux and reboiler are usually run in simulated parallel with one of the stage blocks on a single processor. Unfortunately this means that it is not possible to run a single processor sequential version of the whole column. It is thus difficult to produce fully correct speedup curves.

The separation of the models follows the strictly modular approach which has been taken throughout this thesis. It does however present problems when viewing efficiency. It would be possible to incorporate the reflux, reboiler and stage models together into a single interface, but the work involved to change the underlying communications and management algorithms was not really warranted.

To work around this problem, the results are presented in a slightly different, yet analogous, way. If the code for the reflux, reboiler and stages were merged, the overall effect would be to add the calculation load of two trays to the overall simulation. The end result is essentially the parallelisation of an extended block of stages. If the stage blocks in the test runs are assumed to represent the overall column, the resulting timings can be used if the contribution from the reflux and reboiler can be extracted. This is simply achieved by making sure that the reboiler and reflux programs run on their own processors. Since they use little calculation time compared to the stage programs, the overall execution times will purely represent the time taken for the slowest stage blocks to simulate. This will be valid up to the point where the stage calculation load reaches that of the reflux and reboiler. This occurs at around 1 tray per stage block. With the limited number of processors available for testing this point is never reached. The efficiency of the overall strategy can thus be judged by the efficiency of the stage parallelisation.

Finally, before each set of results was produced, care was taken to ensure that the overall algorithm parallelised cleanly with the models used. Initial testing has made sure that single processor runs take the same number of iterations
to converge as multi-processor runs. The modularisation approach can cause problems with parallel efficiency if anomalies in the modelling equations are not picked up and correctly managed.

**Parallelisation Results**

The main factor affecting overall efficiency of a parallel program is the calculation to communications ratio. The higher this ratio the more efficient the parallel simulator will run. For a given simulation problem, the amount of calculation required is fixed. Thus, the only way of maximising the calculation to communication ratio is by minimising the overall communications overhead and making sure that the work load is evenly distributed.

The design of PDist has focussed greatly on making sure that the communications overhead is as low as possible for a well balanced system. To examine the effectiveness of the overall strategy, two sets of simulation results are presented. Each set contains the simulation execution times for the same dynamic simulation problem on a number of different processors. The main difference between the two simulations are that one uses simple v.l.e and the other uses complex v.l.e provided by PPDS [88]. When compared, the two give a good impression of the variation of the parallelisation approach for both simple and relatively complex problems. The actual simulation is of a 40 tray Methanol/Ethanol/Water column perturbed with a feed composition drop.

The timings obtained from a number of test runs are shown graphically in Figures 6.1 and 6.2. Both graphs show the overall execution time of the simulation versus the number of processors used. In each graph, the execution times obtained with solution transfer on and off are shown. The timings curve expected for linear speedup, 100% efficiency, is also shown on both graphs. To accompany these, Figure 6.3 shows a graph of the parallel efficiencies calculated from the two sets
Figure 6.1: Timings Plot for Simulation With Simple Physical Properties

The timings curves obtained with solution transfer communications off, describe the raw performance of the parallel simulation strategy. The curves with solution transfer communications on, show the effect that adding utility requirement can have on the overall efficiency.

Ignoring solution management effects, the parallelisation strategy appears to work well. In both runs the overall simulation time reduces with added processors. The parallelisation is particularly good for the more complex models, where almost linear speedup is being obtained over the range of processors tested. This is encouraging, since it is the more complex problems for which PDist is targeted at. The fact that the parallel approach also works for simple problems shows that the overall parallel implementation is efficient.
As described in the theory, the implementation methodology was selected for its highly concurrent communication structure and thus potential scalability. The success of the implementation in this area is highlighted in the timings graphs. The distance between the actual timing and the theoretical limit is the overhead incurred in implementing the parallel algorithm. One feature of the results shown, is that the overhead remains roughly constant as more processes are used. This applies equally to both sets of results.

The overall implementation overhead is a product of communication startup costs and imbalances in calculation load distribution. During any simulation, dynamic changes move around in the column. This results in some trays requiring more solution time than others. The parallel implementation used, is designed to be most efficient when the tray calculation loading is equal. When this is not the case, the stage blocks which finish first end up having to wait on the others to catch up. The particular test simulations described here are designed to show
the effect of uneven loading. Figures 6.4 and 6.5 show 3D plots of calculation
time and communications delays versus tray number and simulation time for a
10 processor dynamic simulation.

The simulation carried out was a feed composition drop, where the feed was lo-
cated at tray 20. The effect of this propagates quickly to the surrounding trays.
This shows up in the 3D calculation timings, where the processors above the
feed are initially taking less time to solve than the bottom and top trays. The
uneven loading also moves as the simulation progresses. For the trays where the
calculation load is lowest, the expected result would be an increase in commu-
ications delay due the the required wait for the other trays to catch up. This
is in fact what happens. The 3D communication timings graph shows that the
communication delays peak where the calculation load graph troughs. The inter-
esting point is in the relative size of delay that the uneven loading causes. The
uneven loading time is far bigger than the actual communication implementation.
overhead. For the results shown, as much as 75% of the overall communications overhead is down to uneven tray calculation loading. This is a significant amount.

As well as uneven individual tray calculation load, the efficiency is also affected by the distribution of the actual trays. Unfortunately the number of trays in a column does not always divide equally by the number of processors. In such cases, some stage blocks always have one more tray than the others. This means that linear speedup is only achievable when using certain numbers of processors. Furthermore, as the number of processors increases, so the number that has to be added to obtain any new benefit also increases. e.g For a 40 tray column, the points of exact divisibility are at 1, 2, 4, 5, 8, 10, 20 and 40 processors. For processors used up to 8, the maximum trays on any one processor is always different. However, after 8 the maximum tray count on any of the stages does not change until 10, 15, 20 and 40 processors are used. Between these numbers of processors no parallel benefit will be obtained since there is always one stage block with
Figure 6.5: Graph of Communications Load Distribution During 10 Processor Simulation

the same limiting number of trays. Within this period, the execution time would be expected to increase, if anything, due to slight increases in communications overhead.

The results shown highlight the problem caused by uneven tray distribution. The parallel efficiency oscillates based on the number of processors used, see Figure 6.3. Normally, for a totally scalable problem, this would be expected to simply reduce. The peaks in the efficiency graphs are always found at processor numbers which divide equally into the number of trays. The lack of parallel benefit for higher numbers of processors is also apparent for processor numbers 8 and 9, where at least one stage block always has 5 trays to solve. At 10 processors the trays reduce to 4 and hence a slight parallel benefit is obtained. Although not shown, the next benefit would not be obtained until 15 processors were used. The effect is less distinct for the simpler models since the calculation load of a single tray is much less. This effect is problematic but not disastrous. There are simply
restrictions on the higher numbers of processors that can effectively be used.

The final observation to be made is the detrimental effect which solution transfer communications obviously have on efficiency. This is especially so for simpler problems. The effect is also a cumulative one. Unlike with the simulation critical communications, the solution communication overhead increases as more processors are used. This effect was expected, given that the solution management structure is of the master/slave kind. The graphs shown, do however represent a worst case scenario. For the majority of simulations, highly frequent solution retrieval is only required if fast updating runtime graphics are essential. The problem disappears as soon as solution transfer is made a relatively infrequent event. At present this involves losing all intermediate solutions. However, it is a simple matter to make processors locally store solutions ready for later transfer.

Parallelisation Summary and Conclusions

The parallelisation strategy works well. This is particularly true for problems which are computationally demanding. The more complex each tray calculation is made, the better the overall simulator will perform.

There is very little which can now be done to enhance the overall performance of the simulator. The main remaining problems are associated with utility management and load balancing. Utility management is easily corrected. Load balancing presents a more difficult problem. Besides uneven tray distributions, dynamic changes in columns tend to move around. For this reason some stage blocks are always more computationally demanding than others. This computational requirement also moves around with the column changes. In extreme cases, it has been suggested that tray calculations could be dynamically moved from one stage block to another. However, the time overhead in moving a tray would be significant since all of the tray models state variables would have to move. This
overhead would likely overwhelm any real benefit obtained.

Finally, all of the results shown have been run on a transputer based Meiko Computing Surface. Although the transputer is no longer the fastest processor around, the transputer based system used is still an excellent piece of hardware for exploring parallelism. The newer platforms being produced use much more powerful processors. However, the hardware communication rates have seen similar speed increases and PDist should thus produce similar results on these newer machines.

6.1.5 Summary

The usefulness and efficiency of PDist have been examined. The simulator achieves all of the aims it originally set out to prove. It exploits parallelism, it is extremely usable, it is robust and can be used to tackle real problems.

PDist successfully simulates single columns. One of the most difficult things to cater for has been the types of input that real columns receive from surrounding equipment. This was particularly so for the industrial case study carried out. The simplest way of providing realistic input is to simulate the surrounding equipment. The process network simulator PNet has been designed to make this possible; some tests have been done which include the connection of PDist to other external equipment simulators and these are described in the next section.

6.2 PNet Results

This section describes the initial tests which have been carried out using PNet. The primary aim of the tests has been to make sure that the pipe connection strategy works and that the interface routines for building pipes and simulators
are fully functional. The secondary aim of the tests is to find out how successful the parallelisation strategy works and how useful it is likely to be for the future.

The testing has been carried out by using PNet to create two types of simulator. The simulators created were:

- A sequential modular steady state simulator

  One of the test pipes created is designed to allow PNet to connect steady state simulator modules together in a sequential modular fashion, see Section 5.2.3. For the simulator to work, information must move smoothly around the network in an orderly manner. The pipe mechanism to create this simulator relies heavily on being able to control this information flow and also monitor the convergence of all the pipes in the network. The creation of such a simulator provides an excellent test of the overall communications and connection interface routines.

- A parallel dynamic simulator

  The other pipes created have been designed to allow dynamic simulators to be connected together. For the moment these pipes implement the explicit modular connection strategy, see Section 5.2.3. These pipes, although simple, can be used to find out how well the parallel simulator connection strategy works and where the main inefficiencies lie.

The testing and results obtained are now described.

6.2.1 Steady State Simulator Construction Results

To fully test PNet as a steady state simulator, an inhouse steady state flowsheeting package, ESSPROS [96], has been adapted for use. ESSPROS, is a simple sequential modular flowsheeting package. It contains a number of for-
tran routines for calculating the steady state outputs of various process units given their inputs. A simulation is built by writing a program which contains the required process unit routines in the order of calculation stipulated by the sequential modular approach. Where recycles are present, initial guesses must be provided and the sequence repeated until convergence is obtained. The example ESSPROS programs which have been used in the following description are shown in Appendices D.1.1 and D.1.3.

The main reason for using ESSPROS is that it can be used to build and simulate complex networks extremely quickly. The main purpose of these tests is to make sure that the PNet network works correctly. The calculation load on the actual simulation nodes is irrelevant.

To adapt ESSPROS for use with PNet, the Fortran routines have been replaced by programs. In ESSPROS, each routine's stream connection information is stored in a local database. For the PNet programs, the put and get routines for accessing this database have been replaced with put and get PNet pipe interface routines. For the tests carried out the mixer, splitter, separator, reactor and flash routines have been converted for use. The input description for PNet is built by a simple conversion program. This reads the ESSPROS program, separates the routines and replaces each by a PNet process description. The stream connection names from the routines become pipe names. The various parameters of the ESSPROS routines are passed to the PNet programs using the PNet import mechanism. Appendices D.1.2 and D.1.4 show how the converted ESSPROS examples look.

The two examples shown represent small and highly connected problems. The first example is relatively simple and does not represent any real process. The second example is the flowsheet of an oil separation process on an oil rig. For the tests, the processes used are not particularly relevant. What is important is the number of simulators and pipes required to solve them when using PNet. The two examples have the following topology:
1. 4 process units, 7 pipes of which 1 is a recycle.

2. 14 process units, 24 pipes of which 3 are recycles.

Both ESSPROS and PNet produced identical results for the examples shown. This included the number of iterations taken by each routine or program. PNet also worked well on other ESSPROS problems tested.

In terms of efficiency, the simulators did not behave as badly as expected. In both cases the simulations took only a few seconds, once loaded. The examples were run on a range of processor numbers and also on two hardware platforms: Namely a 12 processor Meiko Computing Surface and a network of SUN4 SPARClstations. For the second example, a single processor run was not possible due to the number of programs involved. It did however run on two workstations and upwards. This fact is not all that worrying. The simulator is designed to use multiple processors and for complex dynamic simulations it is unlikely that all of the simulators could be run on one processor anyway.

Overall, the steady state testing has proved invaluable. The majority of the debugging on PNet was carried out using ESSPROS examples. The simulator and pipe interface routines now work extremely well. The front end loader also worked well. The process structures are being successfully analysed, with all feeds, recycles and product connections being correctly located. In terms of hardware use, RGC worked as expected. ESSPROS simulator programs were run under both CSTools and PVM with no complications. We can conclude that steady state simulation can be performed using the PNet. This is likely to be extremely useful for finding the initial start point for any dynamic simulations being run.
6.2.2 Parallel Dynamic Simulation Results

This section describes the initial results obtained from using PNet for actual dynamic simulation. All of the test runs used have been with the dynamic pipes described in Section 5.2.3.

To use PNet to build a simulator network, some individual process unit simulators are needed. The only relatively complex simulator available to us was PDist. For this reason, PDist has been the main simulator used for the test runs other than some simple mixer and splitter simulators which were created for managing recycles. The PNet connections are not implemented within PDist itself but in the actual PDist modelling interface routines. This way different models with varying connection strategies can easily be catered for.

Two separate simulation examples are presented here. In both cases the pipes which support both the sequential modular steady state and parallel dynamic connection strategies is used. This allows the steady state models in PDist to be used to bring the PNet process examples to steady state before the dynamic simulations are performed.

The two examples used are both distillation train simulations. In each case the components being separated are Methanol, Ethanol, Propanol and Butanol. The first example process is a simple linear train. A flowsheet of this is shown in Figure 6.6. The second example is a similar train with a recycle. The recycle is a potential source of difficulty for the simulations. The example is designed to test how PNet copes with recycles adequately. A flowsheet of this example is shown in figure 6.7. The PNet input descriptions used for the two examples are shown in Appendices D.2.1 and D.2.2.

A number of simulations were carried out using the example networks shown. As with the PDist examples, runs with both simple and complex v.l.e were carried
out. The local number of transputers available was 12. PNet ran each column shown on approximately 4 transputers. As well as PDist code, the transputers also had to run the 7+ pipe managers required for simulator connection.

In both examples the steady state was correctly calculated using the enforced sequential modular connection strategy. The dynamic simulations also worked well. The simulations ran successfully for the full duration. The recycle in the second example caused no problems. In all of the simulations, each column could be interacted with separately and at the end, separate PDist solution viewers were available for analysing the individual column results. The overall connection solutions were also successfully collected and displayed by the PNet solutions manager.

The connection mechanism appears to work very well. A number of changes were
made to the first column in both examples. These changes produced the expected perturbations in the columns farther down stream. No problems were obtained from the connection strategy used. This is probably down to the nature of the inter column links which are dynamically slow. For dynamically faster links, the explicit connection approach may not work so well.

As far as parallelisation efficiency is concerned, the results were better than expected. With the explicit approach used, the overall dynamic simulation is completely parallelised. There is no central control mechanism required, except for solution collection and convergence checking for the steady state calculation. This should make the overall parallelisation efficient. Unfortunately, the loading pattern of the transputers makes it very difficult to estimate the actual efficiency obtained. At the moment PNet simply distributes the simulator programs evenly between the available processors. No account of the likely processor usage of
each program is used. Each column can take varying amounts of time to solve, depending on where and when certain perturbations are taking place.

The strategy used for PNet results in the fact that any simulation can only run as fast as the slowest simulator in the network. The only effective measure of performance is thus to compare the overall simulation time with that of the slowest simulator. In the three column simulations carried out, the overall simulation took almost exactly the same time as the slowest simulation. Hence the parallelisation appeared to work as well as could be expected. To make sure that the actual parallelisation was not extending solution times, the load map was dismantled and the slowest column run using the same processor pattern as in the larger simulation. The times taken for the single column run as compared to the times for a three column simulation were approximately the same. e.g. With simple v.l.e models, a single column took 166 seconds to run as compared to 172 seconds for the three column network. The 6 second difference here being primarily due to the extra communications overhead associated with the PNet master running on the host workstation talking to an extra 8 transputers and slight imbalances in work load between columns throughout the simulation. This overhead should be compared to the saving in execution time of approximately 330 seconds by running the other two columns in parallel.

Overall the parallelisation works well. The biggest problem is obviously going to be with load balancing. The examples described here are relatively simple to load balance, since each column is roughly the same. For more complex process networks with highly variable simulators, a much more sophisticated loader would be required. This loader would need statistics on the average calculation load of each simulator, the communications requirements and a measure of the calculation to communication ratio to help with communication rendezvous optimisation. Work is ongoing in this area by other researchers, some of which used the OCCAM distillation simulator as an example, see Skilling et al [92,97,98].
As well as parallelisation, the other main result is the usability of the simulator. Since each simulator is a program in its own right, building the overall process simulations is simple: it is as easy as drawing a connected set of process units. Once started, manipulating the network simulator is performed by manipulating the individual simulators which make it up. In the examples shown here, each PDist is as usable when connected as when run standalone. The input and output mechanism is identical. The only real difference is that the changes made in one affect all the others. Even if the parallelism produced no benefit, the ease of simulator construction and manipulation which this modular approach provides is, in the view of this author, enough reason to use it.

6.2.3 Summary

PNet has been tested on a number of initial problems. Overall, the modular simulation approach using pipes as simulator connectors has worked well. Both steady state and dynamic problems have been simulated.

The dynamic simulation results have been the most encouraging. Given good load balancing, the parallelisation produces almost linear speedup. For more complex process layouts, this benefit is likely to be much less. The greatest area of work for the future is in working out how best to load process simulators so as to maximise the parallel benefit. The connection approach used should however facilitate this. Given that each simulator is connectable, it is a simple matter to produce a wrapper program for testing out individual simulators. This could be used to fully test a simulator over a range of inputs and also to collect operational statistics as it does so. These statistics could then be used in a loading optimisation package.

Finally, the modular approach has proved extremely useful in its own right for simulator construction and usability. The methodology used is felt to be the way
ahead for the future.
Chapter 7

Conclusions and Future Work

The aim of this work was to investigate the use of MIMD parallel computers for the dynamic simulation of chemical processes. This investigation has been carried out at two levels: the process unit level and the process network level. Throughout a parallel modular approach, rather than a parallel equation based approach, has been used. The modular approach was chosen because of its greater potential for parallelisation, robustness on sequential machines and the ease with which existing and varied simulator code could be combined. The modular approach is also intuitive for the engineer.

7.1 The Modular Approach

The modular approach has been found to be particularly well suited to parallel execution on MIMD machines. Even for simple models, excellent speedups have been obtained using both PDist and PNet. The efficiency of the simulators is primarily down the concurrent structure which the modular approach provides. Almost all communications can be kept as nearest neighbour, with no master/slave bottleneck or routing overhead being present. This has resulted in the parallelisation overhead being dominated by load balance considerations.
Even with PDist, which can be relatively well load balanced by simple geometric division, the load imbalance constituted as much as 75% of the total parallelisation overhead. This load imbalance was also found to become more distinct as models were increased in complexity. Much of the work has focussed on making the communication structures used efficient. For PDist, the results have shown that the communication strategy used is indeed efficient. The results also show that this efficiency is not affected greatly as processor numbers are increased. With PNet, there is still room for improvement with the pipe mechanism. In particular, the feasibility of using one pipe to manage many connections needs to be investigated. Load balancing, on the other hand, has been less well studied. In both PDist and PNet, the loader simply attempts to distribute the simulation programs as evenly as possible between the available processors. For PDist this works well due to the even nature of the problem. For PNet balancing is more complex and it is in this area that future work is required. The actual simulator connection strategy has been proven to work, but it is only through good load balancing that the real benefits of the approach can be obtained.

7.2 Load Balancing

The load balancing problem is common to both parallel modular and parallel equation based methods alike. There is no simple solution in either case. With PDist the load balancing is not particularly uneven. This is due to the nature of distillation, where all the tray calculations are computationally similar. Here, the main imbalances occur when large dynamic changes are taking place in isolated, but changeable, sections of the column. One solution to this is to migrate tray calculations from processor to processor. This would involve having one column section export all of the tray model information from one processor to another. This would incur a large overhead and the load imbalance would have to be significant to warrant such action. The approach is however feasible. For
PNet, individual process block simulators take varying times to solve a given time horizon. This solution time also changes depending on where the disturbances in the process are located. A simulator built out of many simulators can only run as fast as the slowest one over any given time horizon. It is thus important to try and place simple simulations together on single processors, leaving the more complex ones to run on processors of their own. It is not possible to completely load balance a parallel simulator because the problem being simulated is one of fluctuating complexity. It should however be possible to produce a relatively well load balanced simulator if runtime statistics on the various connected simulators can be collected. The modular approach lends itself well to obtaining such statistical information. Any simulator which is adjusted to run in PNet uses pipes to obtain input and send output. It would be a simple matter to produce a test program into which a given simulator could be plugged. The simulator could then be run through a variety of test input scenarios and the statistics associated with solution times collated. The resulting statistics could then become part of the input description to the simulator. At run time the loader could analyse the statistics for each simulator being loaded and attempt to produce as even a program loading as possible on the available hardware. An exceptionally clever loader could even analyse the changes which were to be made in a simulation and attempt to isolate those simulators which are going to go through the biggest dynamic changes. Once the simulators have been loaded there is little which can be done if the load balancing for some reason becomes inefficient. Again a possible solution could be to move simulators from one processor to another. Unfortunately migrating running programs is extremely difficult, expensive to initiate and not a standard operation provided by any hardware and software suppliers. If the load balancing problem can be solved, which this author believes to be possible via the method outlined above, the modular approach can be extremely efficient. Simulation results from PDist have shown the efficiencies which can be obtained given reasonable load balancing.
7.3 Robustness

The parallel modular approach was also expected to be robust. From test cases run using PDist, it does appear to retain the robust characteristics of its sequential counterpart. The single most significant contribution to this robustness is the fact that each individual simulation module can be programmed to use the most robust approach to solving its local problem. This includes choice of integration method, time step, physical properties, solution method and general discontinuity management. The industrial case study models showed the benefit of this, where the problem was solved by combining information and solution methods from many sources into specialised modules. The onsite modelling attempt of the company was only finally made to work once the results from PDist were used to provide initial solution estimates for their column simulator.

Overall, the parallel modular approach is as robust as the modules which make it up. To make sure that the individual modules are robust, similar tests to those suggested for collecting execution time statistics could be used, with input changes designed to test the robustness of the module rather than its execution efficiency.

7.4 Flexibility

Given that parallel modular approach can offer a robust mechanism for building a large simulator out of many smaller ones, it could also be used as a mechanism for breaking up large equation based simulations. Instead of using a single solver, the equations could be split, based on process topology, and a separate solver used on each subset of equations. In this case the PNet pipes would be the partition points between sets of equations. By partitioning the equation based simulators in this way they would become flexible and interchangeable. Many
industrial users find that they can only use certain commercial packages to solve certain process sections. Unfortunately this means that if the two process sections belong to the same chemical process, the whole process can only be simulated by joining the simulators together. This is not usually possible when different packages are involved. The parallel modular approach used with PNet offers a solution to this. Any simulator which is connectable via pipes can connect to any other, no matter what it is like internally. Although the modularisation of equation based simulators offers the benefit of efficient parallelisation and ease of connection to other packages, it goes directly against the grain of the perceived advantages of modular versus equation based. It is quite possible however that the argument has gone on for so many years purely because the real solution lies somewhere in between. From a recent keynote speech by Herbert Britt of Aspen Technology Ltd [99], it appears that such an approach is now being taken with the equation based simulator SPEEDUP for simulating complete plants in very great detail. SPEEDUP represents the most significant implementation to date of the equation based approach to simulation. For very complex problems SPEEDUP is to be split into connected sections, each running on a separate workstation. The company is aiming more towards workstation technology than supercomputers of the Cray variety. PNet has already shown that the modular approach works well, and it is encouraging to note that the biggest simulator company now feels that our approach taken is the solution to obtaining the processing power required to run an equation based simulator on many processors.

7.5 Usability

As well as exploiting parallelism for processing power reasons, this work has attempted to show that, within a parallel environment, simulators can be as usable as their sequential counterparts. We have shown the effectiveness of using specialised programs for specialised tasks. PDist uses this effectively by having the
simulator programs completely separate from the graphics and interaction utility programs, the only connection being through data communication. By standardizing on this data format, programs can relay requests simply and efficiently to each other. Software development is also simplified. The software writer's task is reduced from being one where new and existing codes must be closely integrated, to one where the new code requires only to understand the communication protocol with its associated programs. This in turn allows many different solutions to a single problem. Changing a single unit simulator, interaction or graphics mechanism involves changing a single program. Column models are built into reflux, stage and reboiler programs. Different columns are simulated by using different versions of these programs. Changing a model requires no recompilation or equation reordering. Also, although the modelling programs are being changed, the support tools remain the same since they always talk to the simulation modules using the same protocol.

7.6 User Interfaces

The final topic in this work has been that of providing input, output and interaction for the simulators, again in a modular way. For all simulators loaded with PNet, each is allowed its own input and output mechanism. It is thus possible to write very specialised graphical interfaces which are tailored specifically to the process unit or section being simulated. This has been demonstrated with PDist, where a comprehensive input and output representation is used. The input and output data formats provide a mechanism for fully defining the process being simulated, the models, the changes required during execution and the structure of the output along with a number of useful solution display recommendations. The formats were designed only to test the concept, but work well. Having the simulation input and output descriptions in a standard form provides a convenient mechanism for informing many different programs about a given problem.
The programs can then be designed around the data format rather than the simulation problem itself. An example of this is the PDist solutions viewer. Once it has been given the PDist model output description it automatically tailors itself to present the solutions for a particular set of column models in the exact form and grouping recommended by the modeller. When many PDist models are together, each PDist viewer is the same program but may appear differently depending on the specific models being used. For simulation input, interaction and solutions viewing, the future definitely lies in this direction. The hard task is to find the correct data representation which can describe all of the various fixed and variable attributes of the problem being simulated. The representation of such data is now being investigated by such projects as épée [94], of which this author is a part, and STEP [100]. The épée project is particularly concerned with process engineering.

7.7 Summary

The parallel modular approach shows significant potential. This work has shown that the approach is feasible, maps efficiently to parallel processing, can be applied to real problems and is above all robust. It is the view of this author that the approach used by PNet, rather than a solely equation based approach, represents the way forward. It is felt that the best role for equation based simulators is in the solution of the individual connected simulators.

7.8 Future Work

The simulators produced have been designed to show the benefits of using a parallel modular approach to dynamic simulation. With PDist, the simulator has been taken to a level were there is little in the way of future work required.
The simulator embodies all of the original concepts envisaged and has been used to simulate real problems. The main area where work is required is in the simulator input and output. However, this work applies to simulation in general and not just PDist.

In contrast, PNet is in a less well developed state. The areas where future work is required are:

- Development of control connections

  The only connection types currently available are material connections. It should be possible to manage control connections in a similar way. The control data managers would be used to take control analysis readings from simulators and produce control actions in return. The managers would more closely resemble simulators than pipes.

- Full support for implicit integration strategies

  The convergence control mechanism in PNet does not yet fully support implicit integration. It is important that PNet support this feature to help provide a more robust connection mechanism.

- The production of a statistics based load balancing mechanism

  For PNet to be efficient on parallel hardware, it must be well load balanced. A mechanism is required which can provide this load balancing. To do this it will have to utilise statistical information on the processing requirement of each simulator being connected. A mechanism for obtaining these statistics was outlined in Section 7.2.

- The connection of existing industrial simulators

  The real test of PNet will be to use it to connect the simulators used by industry. If all of the above changes can be made, it should be possible to build very large and complex simulations out of varied and existing codes.
• The data structures for defining the input and output to simulators

The description of the input and output of simulations needs to be addressed. It is only through standardisation on the description of the overall problem and the models used to solve it that the approach used by PNet can be automated. The épée [94] project is already tackling this problem.

7.9 The Future

The future is a parallel one. The technology which was once so specialised is slowly beginning to merge with more conventional hardware. The same processors used to build workstations are now being used to build parallel machines. Workstations are also being produced with multiple processors as standard. The point will come when many single or multiple processor workstations will be connectable to produce an efficient MIMD style parallel machine. It is only slow hardware and software networking which is preventing this from happening now. Through packages like PVM [93] this work has already shown that workstation networks can be easily programmed in a manner analogous to parallel machines. For the engineer, the ability to use standard equipment as a parallel resource will offer a cost effective way of obtaining raw processing power.

In terms of utilising this processing power there is still a lot of research required. Most work to date has been theoretical. There has been very little in the way of actual implementation. For the majority of problems, the main requirement is still for a good general parallel equation solver. It is only in areas such as dynamic simulation, were there is obvious internal parallelism, that specialised solution approaches become possible. In the end, the effective use of parallelism is going to require both standard and specialised approaches. The work on PNet has been built around this concept. At the moment the specialised approaches are being paid the greatest attention. The main drive for the future should
be towards developing standard techniques for equation solving. The parallel hardware is ready to use. It is up to industry to decide if they want to exploit it.
Appendix A

PDist Dynamic Simulation Model Descriptions

This appendix contains the modelling equations and solution methods produced by this author for testing PDist. Together the models described can be used to simulate both conventional columns and the reactive/azeotropic column.

The symbols used in the model descriptions which follow are fully defined in Appendix B.

A.1 General Tray Model Description

Two different models have been produced: One for steady state simulation and one for dynamic simulation. Figure A.1 below shows a general distillation tray.

The unsteady state material balance for tray $n$ is:

$$\frac{dM_n}{dt} = F + L_{n+1} + V_{n-1} - L_n - V_n$$  \hspace{1cm} (A.1)
The unsteady state component balance for component $i$ on tray $n$ is:

$$\frac{dM_{n,i}}{dt} = Fx_F + L_{n+1}x_{n+1,i} + V_{n-1}y_{n-1,i} - L_nx_{n,i} - V_ny_{n,i}$$  \hspace{1cm} (A.2)

We want to try and express everything in terms of as few unknowns as possible. The liquid and vapour flowrates from the tray can be calculated from tray hydraulics. This is discussed in the next section. We can also express $y_{n,i}$ in terms of $x_{n,i}$ from simple vapour/liquid equilibrium:

$$y_{n,i} = K_{n,i}x_{n,i}$$  \hspace{1cm} (A.3)

In most trays, vapour equilibrium is never reached. An alternative way to express $y_{n,i}$ is to use the definition of the Murphree Efficiency:

$$y_{n,i} = eff(K_{n,i}x_{n,i} - y_{n-1,i}) + y_{n-1,i}$$  \hspace{1cm} (A.4)

$K_{n,i}$ is evaluated from a vle model.

$\frac{dM_{n,i}}{dt}$ can also be expressed in terms of $x_{n,i}$:
\[
\frac{dM_{n,i}}{dt} = \frac{d}{dt} [M_n x_{n,i}] = M_n \frac{dx_{n,i}}{dt} + x_{n,i} \frac{dM_n}{dt}
\]
(A.5)

\[\frac{dM_n}{dt}\] and \[\frac{dx_{n,i}}{dt}\] can be integrated from known conditions \(M_n^0\) and \(x_{n,i}^0\) at time \(t\) to the unknown conditions at time \(t + \delta t\). In general any integration procedure may be used. However, because of the potential stiffness of the problem and the need to ensure robustness of the solution, a method which guarantees numerical stability is required. The simplest method is the backward implicit Euler formulation, from which Equation A.5 becomes:

\[
\frac{dM_{n,i}}{dt} = M_n \left( \frac{x_{n,i} - x_{n,i}^0}{\delta t} \right) + x_{n,i} \frac{dM_n}{dt}
\]
(A.6)

In this equation \(\frac{dM_n}{dt}\) can be calculated from Equation A.1. The value of \(M_n\) can be calculated either by simple integration of \(\frac{dM_n}{dt}\) or from tray hydraulics. In the models described here tray hydraulics are used to determine \(M_n\).

We can now express Equation A.2 in terms of two unknowns. The unknowns are \(x_{n,i}\) and \(K_{n,i}\). Overall this gives us \(C\) equations with \(2C\) unknowns, where \(C\) is the number of components. However, we can express \(K_{n,i}\) in terms of a base \(K\) value \(K_B\) and the relative volatility of the component, \(r\nu_{n,i}\):

\[
K_{n,i} = K_B r\nu_{n,i}
\]
(A.7)

If the relative volatility is assumed constant over the solution we now have an equation in just \(x_{n,i}\) and \(K_B\). Overall we have \(C\) equations and \((C+1)\) unknowns. To solve the system we need another equation. This equation is simple and is based on the principle that the liquid and vapour molefractions must add up to one. The best way of expressing this has been found to be:
To solve the system of equations we could use a standard nonlinear equation solver. However for this system of equations there is a simpler method which can be used. By rearranging equation A.2 we can obtain an explicit equation for $x_{n,i}$ in terms of $K_B$. This has two forms: one for the steady state and one for the dynamic state. For the steady state case there are no differential terms. This gives us:

$$x_{n,i} = \frac{Fx_F + L_{n+1,i}x_{n+1,i} + V_{n-1}y_{n-1,i} + V_ny_{n-1,i}(eff - 1)}{L_n + V_nr_{n,i}K_{B eff}}$$  \hspace{1cm} (A.9)

For the dynamic case we get:

$$x_{n,i} = \frac{Fx_F + L_{n+1,i}x_{n+1,i} + V_{n-1}y_{n-1,i} + \frac{M_nz_{n,i}^0}{\delta t} + V_ny_{n-1,i}(eff - 1)}{L_n + V_nr_{n,i}K_{B eff} + \frac{dM_n}{\delta t} + \frac{M_n}{\delta t}}$$  \hspace{1cm} (A.10)

For both cases $y_{n,i}$ can be calculated from Equation A.4. Since Equations A.9 and A.10 are explicit, it is simple to find the overall solution for any given value of $K_B$. All that is required is a suitable method for locating the correct value of $K_B$ which satisfies the constraint Equation A.8.

The method used by the routines to locate the correct value of $K_B$ is the Regula-Falsi method. This method involves making a guess for $K_B$. From this guess the liquid and vapour molefractions can be calculated using the above explicit equations. The constraint Equation A.8 can then be calculated. From the resulting value and some old guesses of $K_B$ the Regula-Falsi is used to make a new guess at the correct value for $K_B$. This continues until the constraint equation is satisfied to a given tolerance.
A.2 Tray Liquid and Vapour Flowrate Models

This section describes the models used to determine the liquid and vapour flowrates from a tray.

Firstly the liquid flowrate model. As before the overall tray mass balance is:

\[ \frac{dM}{dt} = F + L_{n+1} + V_{n-1} - L_n - V_n \]  \hspace{1cm} (A.11)

The simplest hydraulic model uses the relationship:

\[ L_n \approx \frac{1}{\tau_n} M_n \]  \hspace{1cm} (A.12)

where \( \tau_n \) is a characteristic of the tray and fluid. See the section A.3 on tray hydraulics.

From this we can say:

\[ \frac{dM}{dt} \approx \tau_n \frac{dL_n}{dt} \]  \hspace{1cm} (A.13)

By substituting Equation A.11 in Equation A.13 and again applying backward implicit Euler an expression for \( L_n \) is obtained:

\[ L_n = \frac{(F + L_{n+1} + V_{n-1} - V_n)\delta t + L_n\tau_n}{(\tau_n + \delta t)} \]  \hspace{1cm} (A.14)

\( \tau_n \) and \( M_n \), the hydraulic tray constant and the tray holdup, are determined from tray hydraulics. This is described in the next section.
For the steady state model no hydraulics are used. Thus the steady state equivalent of Equation A.14 is:

\[ L_n = L_{n-1} + qF \]  
(A.15)

The vapour model is very simple. For both steady state and dynamic models it takes the form:

\[ V_n = V_{n-1} + (1 - q)F \]  
(A.16)

### A.3 Tray Hydraulic Models

This section describes the hydraulic models used to determine the tray holdup and tray hydraulic constant.

The hydraulic equations which have been used are those used by Gani [101, 102]. The tray hydraulic equations are used to determine \( \tau_n \), the hydraulic tray constant, and \( M_n \), the tray holdup.

Figure A.2 below shows a typical distillation tray and its hydraulic features.

The holdup \( M_n \) of a tray can be calculated from:

\[ M_n = \frac{h_l A_n \rho_n L}{M_n} \]  
(A.17)

where \( h_l \) is the head due to the liquid on the tray. This can be calculated from the following equation:
Figure A.2: Figure Showing Hydraulic Features of a Distillation Tray

\[ h_l = Q_n(h_w + h_{ow}) \]  \hspace{1cm} (A.18)

\( Q_n \) is the aeration factor for the tray. The height over the weir, \( h_{ow} \), can be calculated using various correlations. The correlation used in this model is:

\[ h_{ow} = 0.26\Psi^{-0.37}(\frac{q}{W_l})^{0.87} \]  \hspace{1cm} (A.19)

\( q \) is the volumetric liquid flowrate, \( W_l \) is the weir length and \( \Psi \) is a flow ratio group defined by the following equation:

\[ \Psi = \left( \frac{q}{W_l} \right) \left( \frac{1}{U_g} \right) \left( \frac{\rho_n^L}{\rho_n^V} \right) \]  \hspace{1cm} (A.20)

\[ q = \frac{L_n M_n^{wt} \rho_n^L}{\rho_n^V} \]  \hspace{1cm} (A.21)
Substituting Equations A.18, A.19, A.20 and A.21 into Equation A.17, we obtain an equation of the form:

\[ L_n = \alpha' (M_n - \beta')^{10} \]  

where

\[ \alpha' = \left( \frac{M_n^{\text{wt}}}{Q_n A_o \rho_l} \right)^{10} \left( \frac{0.26 \rho_l}{U_2 \rho_v} \right)^{1.2333} \left( \frac{\rho_l W_l}{M^{\text{wt}}} \right) \]  

and

\[ \beta' = \frac{Q_n h_w A_n \rho_l}{M^{\text{wt}}} \]  

\[ L_n = f(M_n). \]  

By Taylor expansion we get:

\[ L_n = f(M_n^0) + (M_n - M_n^0) f'(M_n^0) + ... \]  

Rearranging:

\[ L_n = [f(M_n^0) - M_n^0 f'(M_n^0)] + M_n f'(M_n^0) \]  

If we take a linear approximation for \( L_n \), i.e

\[ L_n = A + \frac{1}{\tau} M_n^0 \]  

then we can say:
\[
\frac{1}{\tau} = f'(M_n^o) \quad (A.28)
\]

From Equation A.22 we get:

\[
f'(M_n^o) = \frac{10}{3} \alpha' (M_n^o - \beta')^{\frac{7}{3}} \quad (A.29)
\]

This gives us the following expression for the tray hydraulic constant:

\[
\tau = 3 \left( \frac{10}{10\alpha' (M_n^o - \beta')^{\frac{7}{3}}} \right) \quad (A.30)
\]

### A.4 Conventional Reflux Model Description

In this section the simulation models developed for a conventional reflux section are described. The word conventional refers to a reflux section composed of a total condenser and reflux drum. Figure A.3 below shows a conventional reflux section.

![Figure A.3: Conventional Reflux Section](image-url)
Firstly the general component balance model is described. This assumes that the reflux rate, $REF$, and the tops product rate, $TOPS$, have already been calculated. The models used to calculate these flows are dealt with in the next section.

The unsteady state material balance for the section is:

$$\frac{dM_R}{dt} = V_T - REF - TOPS \quad (A.31)$$

The unsteady state component balance for component $i$ is:

$$\frac{dM_{R,i}}{dt} = V_T y_{T,i} - REF x_{R,i} - TOPS x_{R,i} \quad (A.32)$$

At steady state:

$$\frac{dM_{R,i}}{dt} = 0 \quad (A.33)$$

$$V_T = REF + TOPS \quad (A.34)$$

This gives a very simple equation for the steady state model where:

$$x_{R,i} = y_{T,i} \quad (A.35)$$

For the dynamic model $\frac{dM_{R,i}}{dt}$ can be expressed as a function of $x_{R,i}$ using the product rule:
\[
\frac{dM_{R,i}}{dt} = \frac{d}{dt}[M_R x_{R,i}] = M_R \frac{dx_{R,i}}{dt} + x_{R,i} \frac{dM_R}{dt}
\]
(A.36)

\[
\frac{dM_R}{dt} \quad \text{and} \quad \frac{dx_{R,i}}{dt}
\]
can be integrated from known conditions \(M_R^0\) and \(x_{R,i}^0\) at time \(t\) to the unknown conditions at time \(t + dt\). In general any integration procedure may be used. To maintain consistency with the stage models and avoid stiffness problems the backward implicit Euler formulation is again used:

\[
\frac{dx_{R,i}}{dt} = \left( \frac{x_{R,i} - x_{R,i}^0}{dt} \right)
\]
(A.37)

Substituting Equations A.36 and A.37 into A.32 we get an explicit equation for \(x_{R,i}\):

\[
x_{R,i} = \frac{V_{YT,i} + \frac{M_R x_{R,i}^0}{\delta t}}{\frac{dM_R}{dt} + \frac{M_R}{\delta t} + \text{REF} + \text{TOPS}}
\]
(A.38)

Since Equation A.38 is an explicit equation there is no need for a nonlinear solver this time.

Finally the holdup \(M_R\) can be calculated from simple explicit integration using:

\[
M_R = M_R^0 + \frac{dM_R}{dt} \delta t
\]
(A.39)

In the routines used this integration is performed as part of the level control models.
A.5 Conventional Reflux Flowrate Models

In this section the models used to calculate the reflux rate, $REF$, and the tops product, $TOPS$, are described.

In all the models developed so far the tops product is always expressed as a function of the reflux rate. In other words the reflux rate is usually calculated first.

For the steady state model the reflux rate is always set equal to a user defined value. This can similarly be used for the dynamic models. The other option for the dynamic models is to have the rate controlled by a composition controller on the tops product. This model uses a proportional/integral controller to calculate the reflux rate required to try and bring the tops product composition to a given setpoint. The controller models are described later, but for now we can express the reflux rate as:

$$REF = \text{picontroller}(x_{R,i}, x_{SET,i})$$

(A.40)

Once the reflux rate is known the tops product can be calculated. For the steady state model the tops product is obtained by rearranging Equation A.32:

$$TOPS = V_T - REF$$

(A.41)

For the dynamic model the tops product is calculated from the molar holdup of material in the tank. The model used is again a controller model. This model calculates the tops product required to maintain the holdup at a given setpoint. For now we can say:
There is a choice of two control models for the tops product calculation: a proportional/integral model and a proportional controller model.

### A.6 Liquid/Liquid Separator Reflux Model

This section describes the reflux model used for the reactive distillation problem. Figure A.4 below shows a diagram of the reflux section being modelled.

![Diagram of Reactive Distillation Reflux Section](image)

**Figure A.4: Reactive Distillation Reflux Section**

In the reactive distillation problem the vapour stream from the top of the column is cooled. The liquified stream then settles out into two liquid layers. Tops product streams are taken from each layer. The reflux stream is taken from the ester rich layer.

To make the model simpler the refluxed layer is always made the bottom layer. This may not be correct from a density point of view but it means that only one model is required.
The unsteady state material balance for the reflux section is:

$$\frac{dM_R}{dt} = \frac{dM_{R1}}{dt} + \frac{dM_{R2}}{dt} = V_T - REF - TOPS1 - TOPS2 \quad (A.43)$$

The unsteady state component balance for the section is:

$$\frac{dM_{R,i}}{dt} = \frac{dM_{R1,i}}{dt} + \frac{dM_{R2,i}}{dt} = V_T y_{Ti} - REF x_{R1,i} - TOPS1 x_{R1,i} - TOPS2 x_{R2,i} \quad (A.44)$$

As before $\frac{dM_{R,i}}{dt}$ can be expanded by the product rule. This gives:

$$\frac{dM_{R,i}}{dt} = M_{R1} \frac{dx_{R1,i}}{dt} + M_{R1} x_{R1,i} \frac{dM_{R1}}{dt} + M_{R2} \frac{dx_{R2,i}}{dt} + x_{R2,i} \frac{dM_{R2}}{dt} + \quad (A.45)$$

To simplify the model it can be assumed that the holdup in each liquid phase does not change. Hence:

$$\frac{dM_{R1}}{dt} = \frac{dM_{R2}}{dt} = 0 \quad (A.46)$$

As with vapour/liquid equilibrium, the same relationships can be assumed for the liquid/liquid split. Thus $x_{R2,i}$ can be expressed as a function of $x_{R1,i}$ so that:

$$x_{R2,i} = k_i x_{R1,i} = r v_i K_B x_{R1,i} \quad (A.47)$$

The $K$ values for the components can be calculated from an isothermal flash model. This model is explained in the next section.
Finally \( \frac{dx_{R1,i}}{dt} \) can be integrated from the known condition \( x_{R1,i}^0 \) at time \( t \) to the unknown condition at time \( t + \delta t \). Again for compatibility the backward implicit Euler formulation is used.

Substituting Equations A.45 and A.47 into Equation A.44, integrating and rearranging we get an explicit expression for \( x_{R1,i} \) in \( K_B \):

\[
x_{R1,i} = \frac{M_{R1} x_{R1,i}^0 + \frac{M_{R2} r_{V_i} K_B x_{R1,i}^0}{\delta t} + V_{T/Y,i}}{V_{T/Y,i} + \frac{M_{R1}}{\delta t} + \frac{r_{V_i} K_B}{\delta t} + \text{REF} + \text{TOPS1} + \frac{r_{V_i} K_B}{\delta t} \text{TOPS2}}
\]  

(A.48)

The steady state equivalent of this equation is:

\[
x_{R1,i} = \frac{V_{T/Y,i}}{\text{REF} + \text{TOPS1} + \frac{r_{V_i} K_B}{\delta t} \text{TOPS2}}
\]  

(A.49)

As with the stage section models the same constraint equation can be used. This is the equation that states that the molefractions in either phase must add up to one.

\[
\frac{\sum_{i=1}^{C} x_{R2,i}}{\sum_{i=1}^{C} x_{R1,i}} - 1 = \frac{\sum_{i=1}^{C} r_{V_i} K_B x_{R1,i}}{\sum_{i=1}^{C} x_{R1,i}} = 0
\]  

(A.50)

There are now \((C + 1)\) equations and \((C + 1)\) unknowns. The structure of the equations is almost identical to the structure of the stage section tray equations. Again the Regula-Falsi method can be used to find the \( K_B \) value which satisfies the constraint Equation A.50.
A.7 Liquid/Liquid Separator Flowrate Models

This section describes the models used to calculate the reflux rate and tops product for the reactive distillation reflux section.

The flowrate models are very simple. This is because in the component balance equations we have assumed that there is no change in the total holdup of material in the separator. Thus the flowrates can be calculated from the steady state material balance.

\[ V_T - REF - TOPS1 - TOPS2 = 0 \]  \hspace{1cm} (A.51)

The reflux rate is calculated from the reflux ratio. This defines the ratio of the reflux rate to that of the tops product being taken from the same layer. i.e.

\[ RR = \frac{REF}{TOPS1} \]  \hspace{1cm} (A.52)

If \( TOPS2 \) is user defined then by rearrangement an explicit expression for \( REF \) is obtained:

\[ REF = \frac{RR(V_T - TOPS2)}{RR + 1} \]  \hspace{1cm} (A.53)

Finally \( TOPS1 \) can be calculated by rearrangement of equation A.43:

\[ TOPS1 = V_T - REF - TOPS2 \]  \hspace{1cm} (A.54)
A.8 Liquid/Liquid Isothermal Flash Model

This section describes the isothermal liquid/liquid flash model developed to derive the $K$ values used in the liquid/liquid component balance models. Figure A.5 below shows an idealised isothermal flash.

![Isothermal Flash Diagram](image)

Figure A.5: Isothermal Flash

The isothermal flash is used to determine how a given feed of material will split into two liquid phases at a given temperature. The steady state material balance for the separator is:

$$ F = TOPS_1 + TOPS_2 $$  \hspace{1cm} (A.55)

The steady state component balance:

$$ Fx_{F,i} = TOPS_1 x_{1,i} + TOPS_2 x_{2,i} $$  \hspace{1cm} (A.56)

For a liquid/liquid mixture, the chemical potential of each liquid phase must be the same at steady state. The chemical potential for component $i$ is defined as follows:

$$ \mu_i = \left[ \frac{\partial (g_{nT})}{\partial n_i} \right]_{p,T,n,y_i} = \phi(T) + RT \ln(a_i) $$  \hspace{1cm} (A.57)
where \( a_i \) is the activity for the component, \( g \) is the molar Gibbs free energy function for the mixture and \( \phi \) is some function of temperature \( T \). Since the chemical potential of each phase is the same the \( \phi \) terms cancel and we are left with:

\[
a_{1,i} = a_{2,i} \quad \text{(A.58)}
\]

This can then be expressed in terms of activity coefficients:

\[
\gamma_{1,i}x_{1,i} = \gamma_{2,i}x_{2,i} \quad \text{(A.59)}
\]

where

\[
\gamma_i = \frac{a_i}{x_i} \quad \text{(A.60)}
\]

Thus by rearranging we get an expression for \( x_{1,i} \) in terms of \( x_{2,i} \):

\[
x_{2,i} = \frac{\gamma_{1,i}}{\gamma_{2,i}}x_{1,i} \quad \text{(A.61)}
\]

By substituting this into Equation A.56 and rearranging, \( x_{1,i} \) can be expressed as:

\[
x_{1,i} = \frac{F_{x_i}TOPS1}{TOPS1(TOPS1 + \frac{\gamma_{1,i}}{\gamma_{2,i}}TOPS2)} \quad \text{(A.62)}
\]

The activity coefficients can be calculated from the expression which defines the excess chemical potential for a component:
\[ \mu_i^E = \left[ \frac{\partial (g^E n_T)}{\partial n_i} \right]_{p, T, n_j x_j} = RT \ln(\gamma_i) \]  

(A.63)

where \( g^E \) is the molar excess Gibbs free energy function for the mixture. This is the amount by which the Gibbs function of the solution exceeds that of a hypothetical ideal solution of the same composition. \( \gamma_i \) is the activity coefficient for the component.

Since the activity coefficients are strong functions of composition the component balance equations must be solved iteratively. This involves calculating the activity coefficients followed by new estimates for the liquid fractions using Equation A.62. This is repeated until convergence of the liquid fractions is obtained.

There are many different thermodynamic equations for the molar excess Gibbs free energy function. The models developed here use the NRTL equation to calculate the activity coefficients. The NRTL equation is:

\[ \ln(\gamma_i) = \frac{\sum_{j=1}^C \tau_{ji} G_{ji} x_j}{\sum_{i=1}^C G_{ii} x_i} + \sum_{j=1}^C \frac{x_j G_{ij}}{\sum_{i=1}^C G_{ij} x_i} \left[ \tau_{ij} - \frac{\sum_{r=1}^C x_r \tau_{rj} G_{rj}}{\sum_{i=1}^C G_{ij} x_i} \right] \]  

(A.64)

where:

\[ \tau_{ji} = \frac{(g_{ji} - g_{ii})}{RT}; \quad (g_{ji} = g_{ij}, \tau_{ji} \neq \tau_{ij}) \]  

(A.65)

and

\[ G_{ji} = \exp(-\alpha_{ji} \tau_{ji}); \quad (\alpha_{ji} = \alpha_{ij}) \]  

(A.66)

The three adjustable parameters \( (g_{ij} - g_{jj}), (g_{ji} - g_{ii}) \) and \( \alpha_{ij} \) are obtained from
A.9 Conventional Reboiler Model Description

In this section the simulation models developed for a conventional reboiler section are described. Figure A.6 below shows a conventional reboiler section.

\[ \frac{dM}{dt} = L_1 - V - B \]  
(A.67)

The unsteady state component balance for component \( i \) is:

\[ \frac{dM_i}{dt} = L_1 x_{1,i} - V y_i - B x_i \]  
(A.68)

We want to try and express everything in terms of as few unknowns as possible. The boilup rate and bottoms product compositions can be calculated from known data. \( y_i \) can also be expressed as a function of \( x_i \) from simple vapour/liquid equilibrium:
\[ y_i = K_i x_i = r v_i K_B x_i \]  
(A.69)

\( \frac{dM_i}{dt} \) can also be expressed in terms of \( x_i \) and expanded using the product rule:

\[
\frac{dM_i}{dt} = \frac{d}{dt} [M x_i] = M \frac{dx_i}{dt} + x_i \frac{dM}{dt}
\]  
(A.70)

\( \frac{dM}{dt} \) and \( \frac{dx_i}{dt} \) can be integrated from known conditions \( M^0 \) and \( x_i^0 \) at time \( t \) to the unknown conditions at time \( t + \delta t \). In general any integration method may be used. To maintain consistency with the stage and reflux models and to avoid stiffness problems the backward implicit Euler formulation is again used:

\[
\frac{d x_i}{dt} = (\frac{x_i - x_i^0}{\delta t})
\]  
(A.71)

Substituting Equations A.69, A.70 and A.71 into Equations A.68 and rearranging we get an explicit equation for \( x_i \) in terms of \( K_B \). This has two forms: one for the steady state and one for the dynamic state. For the steady state case:

\[
x_i = \frac{L_1 x_{1,i}}{V r v_i K_B + B}
\]  
(A.72)

For the dynamic case we get:

\[
x_i = \frac{L_1 x_{1,i} + \frac{M x_i^2}{\delta t}}{\frac{M}{\delta t} + \frac{dM}{dt} + V r v_i K_B + B}
\]  
(A.73)

As with the stage models the same constraint equation applies here. This constraint is that the sum of the molefractions in each phase must add up to one. This is expressed by:
\[
\sum_{i=1}^{C} \frac{y_i}{x_i} - 1 = \frac{\sum_{i=1}^{C} r_{vi} K_B x_i}{\sum_{i=1}^{C} x_i} = 0
\]  
(A.74)

Since Equations A.72 and A.73 are explicit in terms of \( K_B \), it is simple to find the liquid molefractions given a suitable value of \( K_B \). Thus the same solution method as used for the stage models was used. Here the Regula-Falsi method is used to search for the correct value of \( K_B \) which satisfies the constraint equation A.74.

Finally the holdup \( M \) can be calculated from simple explicit integration as follows:

\[
M = M^o + \frac{dM}{dt} \delta t
\]  
(A.75)

In the routines used for the simulation this integration is performed as part of the level control models.

### A.10 Conventional Reboiler Flowrate Models

In this section the models used to calculate the reboil rate, \( V \), and the bottoms product, \( B \), are described.

In all the models developed so far the bottoms product is always expressed as a function of the reboil rate. In other words the reboil rate is always calculated first.

For the steady state model, the reboil rate is always set by the user. This can also be used for one of the dynamic models. The other dynamic model uses a proportional/integral controller to calculate the boilup rate required to try and bring the bottoms product composition to a given setpoint. The controller models
are described later, but for now we can express the boilup rate as:

\[ V = \text{pcontroller}(x_i, x_{SET_i}) \]  \hspace{1cm} (A.76)

Once the boilup rate is known the bottoms product can be calculated. For the steady state model the bottoms product is calculated by rearranging equation A.67:

\[ B = L_1 - V \]  \hspace{1cm} (A.77)

For the dynamic model the bottoms product is calculated from the molar holdup of material in the reboiler, again using a controller model. The model calculates the bottoms product rate required to maintain the holdup at a given setpoint. The control models are described later. For now we can say:

\[ B = \text{controller}(M, M_{SET}) \]  \hspace{1cm} (A.78)

There is a choice of two control models for the tops product calculation: a proportional/integral model and a proportional model.

### A.11 Reactor/Reboiler Model Description

This section describes the general reactor/reboiler model used for the reactive distillation problem. Figure A.7 below shows a diagram of the reactor/reboiler being modelled.

The unsteady state material balance for the section is:
\[
\frac{dM}{dt} = F + L_1 - V - \text{Stoich}.\text{Rate}.M^{\text{wt}} \tag{A.79}
\]

where \(\text{Stoich}\) is the overall stoichiometry of the reaction, \(\text{Rate}\) is the reaction rate per unit mass and \(M^{\text{wt}}\) is the mass holdup of material in the reactor. In the case of the reactive distillation problem the overall stoichiometry is zero.

The unsteady state component balance for the section is:

\[
\frac{dM_i}{dt} = F x_{Fi} + L_1 x_1,i - V y_i - \text{Stoich}_i \text{Rate}_i M^{\text{wt}} \tag{A.80}
\]

This time \(\text{Stoich}_i\) is the stoichiometry of the individual component with respect to the reaction rate. For example in the reaction:

\[
A + B = C + D \tag{A.81}
\]

the stoichiometry of \(A\) and \(B\) is 1 and the stoichiometry of \(C\) and \(D\) is \(-1\). The overall stoichiometry is 0.

We want to try and express Equation A.80 in as few unknowns as possible. \(y_i\) can be expressed in terms of \(x_i\) using simple vapour/liquid equilibrium:
\[ y_i = K_i x_i = r v_i K_B x_i \]  

(A.82)

The rate term in the equation is usually dependent on composition. This means that we can no longer simply rearrange the equation to allow the Regula-Falsi method to be used. The functions are much more non linear.

The same constraint equation again applies here. This being that the sum of the molefractions in each phase must add up to one. This is best expressed by:

\[ \frac{\sum_{i=1}^{C} y_i}{\sum_{i=1}^{C} x_i} - 1 = \frac{\sum_{i=1}^{C} r v_i K_B x_i}{\sum_{i=1}^{C} x_i} = 0 \]  

(A.83)

Two different approaches have been used to solve the model shown: One uses explicit Euler as the integration method and the other uses backward implicit Euler as the integration method. These approaches are now described:

- Explicit Integration Method

This model uses simple explicit Euler to integrate Equation A.80:

\[ M_i = M_i^0 + \frac{dM_i}{dt} \delta t \]  

(A.84)

The rate and vapour liquid equilibrium variables are all calculated using variables from the previous time step. Once this has been done the new holdup can be calculated by simple summation:

\[ M = \sum_{i=1}^{C} M_i \]  

(A.85)

The new molefractions can then be calculated using:

\[ x_i = \frac{M_i}{M} \]  

(A.86)
The vapour fractions can then be calculated using Equation A.82. Finally the system must be adjusted to satisfy the constraint Equation A.83. This can be performed as before using the Regula-Falsi method to find the correct $K_B$ value.

- Implicit Integration Method

$\frac{dM_i}{dt}$ can be expressed in terms of $x_i$ by rearrangement and expansion using the product rule:

$$\frac{dM_i}{dt} = \frac{d}{dt}[Mx_i] = M\frac{dx_i}{dt} + x_i\frac{dM}{dt}$$  

(A.87)

$\frac{dM}{dt}$ can be evaluated from Equation A.79 and $\frac{dx_i}{dt}$ can be integrated using the backward implicit Euler formulation:

$$\frac{dx_i}{dt} = \frac{(x_i - x_i^\circ)}{\delta t}$$  

(A.88)

Substituting Equations A.82, A.87 and A.88 into Equations A.80 we are left with a set of nonlinear equations. These are the $C$ component balance equations and the constraint equation A.83. These equations can be solved using a standard nonlinear solver. For the models developed here Newton Raphson is used. This is described in section A.19.

The steady state model for this section also requires the use of a nonlinear solver. The equations to be solved are the same as the implicit dynamic ones without the time dependent terms.

### A.12 Reactor/Reboiler Flowrate Models

This section describes the models used to calculate the boilup rate for the reactor/reboiler.
There are two different approaches used to calculate the boilup rate: one for the steady state model and one for the dynamic model. The steady state model uses a simple rearrangement of the steady state material balance:

\[ V = F + L_1 \]  \hspace{1cm} (A.89)

The dynamic model is slightly more complex. If the steady state equation is used the overall change in molar holdup remains constant. This does not mean that the volumetric holdup remains constant. For mixtures whose components have very different molecular weights, slight changes in composition can give large changes in volumetric holdup. Thus in the dynamic model the boilup rate calculated is that which prevents any change in the volumetric holdup in the tank. It essentially models a perfect level controller.

Firstly the volume increase which will be caused by the incoming material must be calculated:

\[ \frac{dM^V}{dt} = \sum_{i=1}^{C} \left( \frac{L_1 x_{1,i} M_{wt_i}}{\rho_i} + \frac{F x_{F,i} M_{wt_i}}{\rho_i} \right) \]  \hspace{1cm} (A.90)

This defines the volume of liquid material which must be removed as vapour. The liquid volume expressed in terms of the vapour variables is:

\[ \frac{dM^V}{dt} = \sum_{i=1}^{C} \left( V_{yi} M_{wt_i} \right) / \rho_i \]  \hspace{1cm} (A.91)

where \( \rho_i \) is still the liquid density of component \( i \).

Equating these equations and rearranging we get an expression for the boilup rate:
\[
V = \frac{\sum_{i=1}^{C} \left( \frac{L_{i}x_{i}M_{u}t_{i}}{\rho_{i}} + \frac{F_{xi}M_{u}t_{i}}{\rho_{i}} \right)}{\sum_{i=1}^{C} \left( \frac{y_{i}M_{u}t_{i}}{\rho_{i}} \right)}
\]  

(A.92)

A.13 Vapour/Liquid Equilibrium Models

The purpose of the vapour/liquid equilibrium models is to calculate the temperature, K values and relative volatilities for given mixtures. This requires a bubble point calculation.

The bubble point of a mixture is the point at which the first bubble of vapour begins to appear. This can be represented by the following equation for ideal mixtures:

\[
\sum_{i=1}^{C} y_{i} = \sum_{i=1}^{C} K_{i}x_{i} = 1
\]  

(A.93)

where \( K_{i} \) can be calculated from Raoult's Law as follows:

\[
K_{i} = \frac{y_{i}}{x_{i}} = \frac{P_{i}^{*}}{P_{T}}
\]  

(A.94)

The vapour pressure for the mixture can be calculated from the Antoine correlation:

\[
\ln(P_{i}^{*}) = A_{i} - \frac{B_{i}}{T + C_{i}}
\]  

(A.95)

where \( A_{i}, B_{i} \) and \( C_{i} \) are the ideal component Antoine coefficients. These are available for most components. Unfortunately the Antoine equation cannot be used
directly for non ideal mixtures since the coefficients are based on pure properties. For non ideal mixtures more rigorous methods are required.

For non ideal mixtures the bubble point can be calculated using equations of state which take into account the affect components have on each other when mixed together. Fortunately these methods have been implemented in various computer packages. For the models developed here the PPDS package is used. The main drawback of these packages is the time required to find a given bubble point. To attempt to reduce the execution time that would be required it was decided that a hybrid of the Antoine equation and the PPDS bubble point method would be used. This involves using the rigorous bubble point calculation to estimate pseudo Antoine coefficients.

In the Antoine equation the coefficient which tends to vary most for different components is the $B$ coefficient. The $A$ coefficient is virtually a global constant and the $C$ coefficient varies only slightly when compared to the variation in $B$. By assuming that the $A$ and $C$ components are fixed, a non ideal estimate for the $B$ coefficient can be calculated:

\[
B'_i = -(\ln(P_i^*) - A_i)(C_i + T)
\]  

(A.96)

where $T$ and $P_i^*$ are calculated from a non ideal bubble point calculation using PPDS. The value of $P_i^*$ being calculated by rearrangement of equation A.94.

During the simulation these pseudo Antoine coefficients can be used instead of the rigorous methods for the vle calculations. This works well as long as the pseudo coefficients are updated once every few iterations.
APPENDIX A. PDIST TEST MODELS

A.14 Proportional/Integral Control Model

The proportional/integral control models are used for composition control of the tops and bottoms streams. The controller proportional part is calculated from the following:

\[ u(t) = \text{bias}(t) + K_c(h(t) - h_{\text{SET}}) \quad (A.97) \]

where \( u \) is the manipulated variable, \( K_c \) is the gain of the controller, \( \text{bias} \) is the current bias, \( h \) is the current value of the controlled variable and \( h_{\text{SET}} \) is the setpoint value. The integral part has the form:

\[ u_{pi}^{(t+\delta t)} = u_{pi}^{(t+\delta t)} + \frac{K_c}{\tau} \int (h - h_{\text{SET}}) dt \quad (A.98) \]

The integral term can be discretised to give an equation in terms of the integration time step \( \delta t \) and an updated bias, where the updated bias contains the integrated error from previous time steps:

\[ u_{pi}^{(t+\delta t)} = \text{bias}^{(t)} + K_c(h(t) - h_{\text{SET}}) + (h(t) - h_{\text{SET}}) \frac{\delta t K_c}{\tau} \quad (A.99) \]

where \( \text{bias}^{(t)} \) is updated at the end of each time step to \( \text{bias}^{(t+\delta t)} \) using a rearrangement of (A.99):

\[ \text{bias}^{(t+\delta t)} = \text{bias}^{(t)} + \frac{K_c}{\tau} (h(t) - h_{\text{SET}}) \delta t = u_{pi}^{(t+\delta t)} - K_c(h(t) - h_{\text{SET}}) \quad (A.100) \]
A.15 Level Control Models

There are two different types of level controllers used: a proportional controller and a proportional/integral controller. For a tank with controlled flowrate \( f \) the overall material balance is:

\[
\frac{dM}{dt} = \text{net flowin} - f \quad (A.101)
\]

where \( \text{net flowin} \) is the sum of all the flows for the unit not including the controlled flow.

This equation can be integrated from known condition \( M^o \) at time \( t \) to the unknown condition at time \( t + \delta t \) using explicit Euler:

\[
M = \frac{dM}{dt} \delta t + M^o \quad (A.102)
\]

Substituting Equation A.101 into this equation and expanding \( f \) using the appropriate controller equations an expression for the holdup \( M \) is obtained:

- **Proportional Level Controller**

\[
M = \frac{M^o + \delta t((\text{net flowin} - \text{bias}) + K_cM_{SET})}{1 + (\delta t K_c)} \quad (A.103)
\]

- **Proportional/Integral Level Controller**

\[
M = \frac{M^o + \delta t((\text{net flowin} - \text{bias}) + K_c(1 + \frac{\delta t}{\tau})M_{SET})}{1 + (\delta t K_c)(1 + \frac{\delta t}{\tau})} \quad (A.104)
\]
A.16 Simulation Input Ramping Models

The ramping of a variable involves changing its value in a linear fashion from an initial condition to a target condition over a specified period. In the current models the feed flowrate and compositions can be ramped. This can be expanded to other variables if required.

For feed flowrates the ramping equation is analogous to the equation for a straight line, where the intercept is taken as the flowrate at the start of the ramp:

\[
\frac{f_{f,\text{i}}}{\text{per}R} = \left(\frac{f_{\text{final}} - f_{\text{init}}}{\text{per}R}\right)t + f_{\text{init}} \tag{A.105}
\]

where \( f_{\text{init}} \) is the starting feed condition, \( f_{\text{final}} \) is the target feed condition, \( \text{per}R \) is the ramp period and \( t \) is the time which has elapsed since the start of the ramp.

For molefractions the ramp is carried out on the individual component molar flowrates. For component \( i \):

\[
f_{x_{f,i}} = \frac{(f_{\text{final}}x_{\text{final},i} - f_{\text{init}}x_{\text{init},i})}{\text{per}R}t + f_{\text{init}}x_{\text{init},i} \tag{A.106}
\]

Substituting equation A.105 into A.106 and rearranging an expression for \( x_{f,i} \) is obtained:

\[
x_{f,i} = \frac{(f_{\text{final}}x_{\text{final},i} - f_{\text{init}}x_{\text{init},i})}{\text{per}R}t + f_{\text{init}}x_{\text{init},i} \tag{A.107}
\]

The ramping of other variables can be performed in a similar way. The set is limited at present since most of the work involved is concerned with the detection of feed changes and error checking rather than the actual ramping process itself.
A.17 Simulation Input Oscillation Models

The oscillation of variables involves changing its value sinusoidally between an initial and target condition with a given period. In the current models the feed flowrate and molefractions can be oscillated. This can be expanded to other variables if required.

Unlike with ramping the reference point used for all calculations is the mean of the start and finish condition. A sine wave is then generated around this value with maximum amplitude equal to half the difference between the initial and target conditions. Figure A.8 below shows the approach used:

\[ f = \left( \frac{f_{\text{init}} + f_{\text{final}}}{2} \right) + \left( \frac{f_{\text{final}} - f_{\text{init}}}{2} \right) \sin(\theta) \]  

(A.108)

For feed flowrates the oscillation equation is:

Figure A.8: Oscillation Approach Used
\[
\theta = \frac{t}{\text{perc}} 2\pi + \frac{3\pi}{4} \quad \text{(A.109)}
\]

The last term ensures that at time zero the flowrate is set equal to the initial condition. As before the oscillation of molefractions is calculated from the oscillation of component molar flowrates. For component \( i \):

\[
f_{xf,i} = \left( \frac{f_{\text{init}}x_{\text{init},i} + f_{\text{final}}x_{\text{final},i}}{2} \right) + \frac{f_{\text{final}}x_{\text{final},i} - f_{\text{init}}x_{\text{init},i}}{2} \sin(\theta) \quad \text{(A.110)}
\]

Substituting equation A.108 into A.110 and rearranging an expression for \( x_{f,i} \) is obtained:

\[
x_{f,i} = \frac{(f_{\text{init}}x_{\text{init},i} + f_{\text{final}}x_{\text{final},i})}{2} + \frac{(f_{\text{final}}x_{\text{final},i} - f_{\text{init}}x_{\text{init},i})}{2} \sin(\theta) \quad \text{(A.111)}
\]

The oscillation of other variables can be performed in a similar way. The set is limited at present since most of the work involved is concerned with the detection of feed changes and error checking rather than the actual oscillation process itself.

**A.18 The Regula-Falsi Method**

The Regula-Falsi method is a root finding method for nonlinear equations. It is generally used for functions which are smooth near the root. The method works by assuming that the function is approximately linear in the region of interest and the next estimate of the root is taken as the point where the line between the last two function values crosses the axis. Figure A.9 shows how the method is used to find a new estimate at the root.
Figure A.9: The Regula-Falsi Method

The updated value \( x_{\text{new}} \) can be calculated using similar triangles where:

\[
\frac{f_2}{(f_2 - f_1)} = \frac{x_2 - x_{\text{new}}}{x_2 - x_1} \tag{A.112}
\]

Rearranging this we get:

\[
x_{\text{new}} = \frac{f_1 x_2 - f_2 x_1}{f_1 - f_2} \tag{A.113}
\]

The order of convergence of the Regula-Falsi method is approximately the “Golden Ratio” 1.618.

A.19 Newtons Method

Newtons method is used here to solve systems of nonlinear equations. It is especially suited for systems where the neighbourhood of a root can be identified. This makes it ideal for solving component balance equations where the molefractions have known bounds.
A typical problem gives $N$ functional relationships to be zeroed:

$$f_i(x_1, x_2, \ldots, x_N) = 0 \quad i = 1, 2, \ldots, N \quad (A.114)$$

Let $\mathbf{X}$ denote the entire vector of values $x_i$, then in the neighbourhood of $\mathbf{X}$, each of the function values $f_i$ can be expanded by Taylor series:

$$f_i(\mathbf{X} + \delta \mathbf{X}) = f_i(\mathbf{X}) + \sum_{j=1}^{N} \frac{\partial f_i}{\partial x_j} \delta x_j + O(\delta \mathbf{X}^2) \quad (A.115)$$

By neglecting terms of order higher than $\delta \mathbf{X}^2$, a set of linear equations for the corrections $\delta \mathbf{X}$ that move each function closer to zero simultaneously are obtained:

$$\sum_{j=1}^{N} \alpha_{ij} \delta x_j = \beta_i \quad (A.116)$$

where

$$\alpha_{ij} = \frac{\partial f_i}{\partial x_j} \quad \beta_i = -f_i \quad (A.117)$$

This set of equations can be solved using a linear solver. The corrections can then be added to the solution vector:

$$x_i^{\text{new}} = x_i^{\text{old}} + \delta x_i \quad i = 1, \ldots, N \quad (A.118)$$

This process can be be repeated until convergence is obtained.
Appendix B

Symbols Used In PDist Model Descriptions

B.1 General Model Symbols

\(a_i\) - The activity of component \(i\).
\(a_{1,i}\) - The activity of component \(i\) in the bottom layer of a two phase mixture.
\(a_{2,1}\) - The activity of component \(i\) in the top layer of a two phase mixture.
\(a_{ij}\) - An NRTL parameter.
\(c_{ij}\) - An NRTL parameter.
\(A_i\) - Antoine A coefficient for component \(i\).
\(B\) - The total molar bottoms product.
\(b_{ij}\) - An NRTL parameter.
\(B_i\) - Antoine B coefficient for component \(i\).
\(B_i'\) - Pseudo non-ideal Antoine B coefficient for component \(i\).
\(C_i\) - Antoine C coefficient for component \(i\).
\(C\) - The number of components.
\(eff\) - Tray efficiency.
\(f\) - Total molar feed flowrate.
\(F\) - Total molar feed flowrate.
$g$ - The Gibbs molar free energy function.
$g^E$ - The excess Gibbs molar free energy function.
$g_{ij}$ - An NRTL parameter.
$G_{ij}$ - An NRTL parameter.
$K'$ - The current value of the equilibrium ratio.
$KE$ - The equilibrium constant.
$k_f$ - The forward reaction rate constant.
$K_i$ - $K$ value for component $i$.
$K_B$ - Root $K$ value of the base component.
$K_{n,i}$ - $K$ value of component $i$ on tray $n$.
$L_1$ - The total molar liquid flowrate leaving the bottom of the column.
$L_n$ - Total molar liquid flowrate leaving tray $n$.
$L_{n+1}$ - Total molar liquid flowrate entering tray $n$ from above.
$M$ - Total molar holdup.
$M^o$ - Total molar holdup at last time step.
$M_R$ - Total molar holdup of material in the reflux decanter.
$M_{R1}$ - The total molar holdup of the bottom layer in the reflux decanter.
$M_{R2}$ - The total molar holdup of the top layer in the reflux decanter.
$M^V$ - The volumetric holdup of material.
$mwt_i$ - The molecular weight of component $i$.
$M^m_{n}$ - Total mass holdup of material on tray $n$.
$n_i$ - The number of moles of component $i$.
$n_T$ - The total number of moles.
$P_i^*$ - Vapour pressure of component $i$.
$P_T$ - Total pressure.
$q$ - $Q$ value for feed.
$R$ - The Gas constant.
$Rate$ - The reaction rate.
$REF$ - The molar reflux rate.
$RR$ - The reflux ratio.
APPENDIX B. SYMBOLS USED IN PDIST MODEL DESCRIPTIONS

- Relative volatility of component \( i \) on tray \( n \).

- The overall stoichiometry of a reaction.

- The reaction stoichiometry with respect to component \( i \).

- Temperature.

- The temperature in kelvin.

- The molar tops product rate.

- The molar tops product offtake from the bottom layer.

- The molar tops product offtake from the top layer.

- Total molar vapour flowrate leaving tray \( n \).

- Total molar vapour flowrate entering tray \( n \) from below.

- Molefraction for component \( i \) in feed.

- Molefraction of component \( i \) in feed.

- Liquid molefraction of component \( i \) on tray \( n \).

- Molefraction of component \( i \) on tray \( n \) at last time step.

- Liquid molefraction of component \( i \) in liquid entering tray \( n \).

- The liquid molefraction of component \( i \) in the reflux drum.

- The liquid molefraction of component \( i \) in the bottom layer of the reflux decanter.

- The liquid molefraction of component \( i \) in the top layer of the reflux decanter.

- Vapour molefraction of component \( i \).

- Vapour molefraction of component \( i \) on tray \( n \).

- Vapour molefraction of component \( i \) in vapour entering tray \( n \).

- The vapour molefraction of component \( i \) in the vapour stream exiting the column.

- Rate of change of liquid flowrate leaving tray \( n \).

- Rate of change of molar holdup.

- Rate of change of total molar holdup on tray \( n \).

- Rate of change of molar holdup of component \( i \) on tray \( n \).

- The rate of change of molar holdup of component \( i \) in the reflux drum.

- Rate of change of total molar holdup in the reflux decanter.

- An NRTL parameter.
APPENDIX B. SYMBOLS USED IN PDIST MODEL DESCRIPTIONS

\[ \gamma_{1,i} \rightarrow \text{The activity coefficient of component } i \text{ in the bottom layer of a two phase mixture.} \]

\[ \gamma_{2,1} \rightarrow \text{The activity coefficient of component } i \text{ in the top layer of a two phase mixture.} \]

\[ \delta t \rightarrow \text{Time step} \]

\[ \mu_i \rightarrow \text{The chemical potential of component } i. \]

\[ \mu_i^E \rightarrow \text{The excess chemical potential of component } i. \]

\[ \rho_n^L \rightarrow \text{Density of liquid leaving tray } n. \]

\[ \tau_{i,j} \rightarrow \text{An NRTL parameter.} \]

\[ \phi(T) \rightarrow \text{A function of } T \text{ used in chemical potential equation.} \]

### B.2 Hydraulic Symbols

\[ A_n \rightarrow \text{Active area of tray } n. \]

\[ h_i \rightarrow \text{Height of liquid on tray } n. \]

\[ h_{aw} \rightarrow \text{Height of the liquid over the tray weir.} \]

\[ h_w \rightarrow \text{Height of the tray weir.} \]

\[ Q_n \rightarrow \text{Aeration factor for tray.} \]

\[ U_n \rightarrow \text{The vapour velocity based on active area.} \]

\[ W_i \rightarrow \text{The length of the tray weir.} \]

\[ \rho_n^L \rightarrow \text{Density of liquid leaving tray } n. \]

\[ \rho_n^V \rightarrow \text{Density of vapour leaving tray } n. \]

\[ \alpha_{i,j} \rightarrow \text{Variable used for hydraulics.} \]

\[ \beta_i \rightarrow \text{Variable used for hydraulics.} \]

\[ \tau_n \rightarrow \text{Hydraulic time constant for tray } n. \]

\[ \Psi \rightarrow \text{Hydraulic variable representing an empirical relationship.} \]
B.3 Control Symbols

\( bias(t) \) - Controller bias at time \( t \).
\( bias(t+\delta t) \) - Controller bias at new time \( t + \delta t \).
\( h \) - Measured variable.
\( h(t) \) - Measured variable value at time \( t \).
\( h_{SET} \) - Setpoint value for the measured variable.
\( K_c \) - Controller gain.
\( M_{SET} \) - Setpoint total molar holdup.
\( net\,flow\,in \) - Net sum of all flows except manipulated flow.
\( u_p^{(t+\delta t)} \) - New manipulated variable value calculated by a proportional controller.
\( u_{pi}^{(t+\delta t)} \) - New manipulated variable value calculated by a proportional/integral control mode.
\( \tau \) - Time constant for proportional/integral controller.

B.4 Ramping and Oscillation Symbols

\( f_{init} \) - Initial value for total molar feed flowrate.
\( f_{final} \) - Target value for total molar feed flowrate.
\( per_R \) - Ramping period.
\( per_O \) - Oscillation period.
\( t \) - Time since start of ramp/oscillation.
\( x_{final,i} \) - Target value for feed molefraction of component \( i \).
\( x_{init,i} \) - Initial value for feed molefraction of component \( i \).
\( \theta \) - Phase term for oscillation mode.
\( \pi \) - Physical constant \( \pi = 3.141592 \ldots \).
B.5 Numerical Method Symbols

$f()$  - A function $f$.
$f'(())$ - The derivative of a function $f$.
$r$   - Relative linearity of function at local point.
$x_1$ - First guess for root in secant method.
$x_2$ - Second guess for root in secant method.
$x_{new}$ - New guess for root in secant method.
$x_{now}$ - New guess for root in secant method.
$X$   - Vector of all values of $x_i$.
$\frac{\partial h_i}{\partial x_j}$ - Rate of change of function $i$ with respect to variable $j$.
$\delta$ - A finite size.
$\delta X$ - Vector of changes in $X$ when Newton's method is used.
$\delta x_j$ - Finite change in the value of $x_j$. 
Appendix C

PDist Input Variables and Example Input Files

This appendix contains a description of the PDist input variables accessible from the PDist Model Interface routines. The appendix also contains an example PDist input description and PDist output description.

C.1 PDist Input Variables

The following text is taken from the PDist model interface header file pdist_setup.inc. It contains all of the global variables initiated and updated by the PDist environment.

C

File: pdist_setup.inc
Release: 1
Revision: 1.1
Author: R.C. McKinnel
Last Update: 3/22/93
Copyright 3/22/93, Roderick C. McKinnel, All Rights Reserved
Purpose:
APPENDIX C. PDIST INPUT FILES

c PDist input file variables.
c These are setup and manipulated by PDist.
c

*** Overall Column Sizing Variables ***

integer nosofcomps, nosoffeeds, no, basecomp

### nosofcomps - The number of components in the column.
### nosoffeeds - The number of feeds in the column.
### no - The total number of plates in the column.
### basecomp - base component for vle

*** Local Column Sizing Variables ***

integer localbot, nplates, localtop

### localbot - The bottom plate number in modelling section.
### nplates - The number of plates in modelling section.
### localtop - The top plate number in modelling section.

*** Feed Setups ***

double precision f(MAXFEEDS), ftemp(MAXPLATES), fpress(MAXPLATES)
double precision xf(MAXPLATES, MAXCOMPS), q(MAXPLATES)
logical youarefeed(MAXPLATES)
integer feednumber(MAXPLATES)

### f(*) - Vector of feed flowrates per plate.
### ftemp(*) - Vector of feed temperatures.
### fpress(*) - Vector of feed pressures.
### xf(*,*) - Array of feed fractions per plate.
### q(*) - Vector of heat qualities per plate.
### youarefeed(*) - Indicates feed plates. True if plate has a feed.
### feednumber(*) - The number of the feed if any.

*** Simulation Time Variables ***

double precision currenttime, finishtime, dt, histdt

### currenttime - The current simulation time.
### finishtime - The finish time for the simulation.
### dt - The overall simulation time step.
### histdt - The time step for solution storage.

*** Column Variables ***

double precision e(0:MAXPLATES), press(0:MAXPLATES)
APPENDIX C. PDIST INPUT FILES

### e(*)
- The plate efficiencies.

### press(*)
- The pressure on each plate.

### Thermodynamic Variables ***
double precision compmwt(MAXCOMPS), denscomp(MAXCOMPS)
double precision anta(MAXCOMPS), antb(MAXCOMPS), antc(MAXCOMPS)

### compmwt(*)
- The component molecular weights.

### denscomp(*)
- The component densities.

### anta(*)
- Antoine A values for components

### antb(*)
- Antoine B values for components

### antc(*)
- Antoine C values for components

### Tray Hydraulic Variables ***

double precision d, aa, ap, ad, admin, ah, dh, nh, ha, hw, lw, oc, qp

### d
- Diameter of holes.

### aa
- Actual plate area.

### ap
- Active plate area.

### ad
- Area of downcomer.

### admin
- Minimum downcomer area.

### ah
- Area of holes.

### dh
- Diameter of holes.

### nh
- Number of holes.

### ha
- Height of the apron.

### hw
- Height of the weir.

### lw
- Length of the weir.

### oc
- Orifice coefficient.

### qp
- Aeration factor.

### Controller/Model Variables ***

### Reflux ***
integer reflux_controller
double precision reflux_params(MAXCONTROLLERS, MAXCONTPARANS)
integer nreflux_controllers, nreflux_params(MAXCONTROLLERS)

### Tops ***
integer tops_controller
double precision tops_params(MAXCONTROLLERS, MAXCONTPARANS)
integer ntops_controllers, ntops_params(MAXCONTROLLERS)

### Boilup ***
integer reboil_controller
double precision reboil_params(MAXCONTROLLERS, MAXCONTPARANS)
integer nreboil_controllers, nreboil_params(MAXCONTROLLERS)
APPENDIX C. PDIST INPUT FILES

*** Bottoms ***
integer bottoms_controller
double precision bottoms_params(MAXCONTROLLERS, MAXCONTPARAMS)
integer nbottoms_controllers, nbottoms_params(MAXCONTROLLERS)

### reflux_controller - Current reflux controller set.
### reflux_params(*,*) - Reflux controller parameters.
### nreflux_controllers - The number of reflux controllers.
### nreflux_params(*) - The number of parameters per controller.

### tops_controller - Current tops product controller set.
### tops_params(*,*) - Tops controller parameters.
### ntops_controllers - The number of tops product controllers.
### ntops_params(*,*) - The number of parameters per controller.

### reboil_controller - Current boilup rate controller set.
### reboil_params(*,*) - The controller parameters.
### nreboil_controllers - The number of boilup controllers.
### nreboil_params(*) - The number of parameters per controller.

### bottoms_controller - Current bottoms product controller set.
### bottoms_params(*,*) - The controller parameters.
### nbbottoms_controllers - The number of bottoms controllers.
### nbbottoms_params(*) - The number of parameters per controller.

*** User Arguments ***
double precision userargs(MAXARGS)
integer nuserargs

### userargs(*) - The user defined arguments.
### nuserargs - The number of user arguments.

*** Simulation Mode Control ***
logical usingsteadystate

### usingsteadystate - If true. Steady state mode used initially, followed by dynamic mode.

*** Shared Memory Space ***
double precision broadcastbuf(BROADCASTSIZE)

*** Common blocks containing all of the above ***
common/pdistsizes1/nosofcomps, nosoffeeds, no, basecomp
common/pdistsizes2/localbot, nplates, localtop
common/pdistfeed/f, ftemp, fpress, xf, q, youarefeed, feednumber
C.2 PDist Reactive Distillation Input File

This section contains an example PDist input description. The example shown is one used to set up a reactive azeotropic distillation simulation. The actual simulation being set up is explained in the description header.

/*
Purpose:
This is the data file required to perform Case Study 1.
This case study involves investigating the effect of a feed ethanol/water change to the reactor.

The following tables show the initial condition of the reactor feed.
This case is concerned with the Ethanol/Water feed which is set as 96 \%wt initially. It is then ramped to 92 \%wt and run for 3hrs.
Finally it is ramped back up to the initial state of 96\%wt.

Acetic Acid Feed
==========
<table>
<thead>
<tr>
<th>Kg/hr</th>
<th>Kmol/hr</th>
<th>kmol/s</th>
<th>wt%</th>
<th>mol %</th>
</tr>
</thead>
<tbody>
<tr>
<td>Acetic Acid</td>
<td>6424.0000</td>
<td>107.0667</td>
<td>0.0297</td>
<td>1.0000</td>
</tr>
<tr>
<td>Ethanol</td>
<td>0.0000</td>
<td>0.0000</td>
<td>0.0000</td>
<td>0.0000</td>
</tr>
</tbody>
</table>
* Water 0.0000 0.0000 0.0000 0.0000 0.0000
* Butyl Acetate 0.0000 0.0000 0.0000 0.0000 0.0000
* Ethyl Acetate 0.0000 0.0000 0.0000 0.0000 0.0000
* Totals 6424.0000 107.0667 0.0297 1.0000 1.0000
* Ethanol/Water Feed
* ================================
* Kg/hr Kmol/hr kmol/s wt% mol %
* Acetic Acid 0.0000 0.0000 0.0000 0.0000 0.0000
* Ethanol 5198.4000 113.0087 0.0314 0.9600 0.9038
* Water 216.6000 12.0333 0.0033 0.0400 0.0962
* Butyl Acetate 0.0000 0.0000 0.0000 0.0000 0.0000
* Ethyl Acetate 0.0000 0.0000 0.0000 0.0000 0.0000
* Totals 5415.0000 125.0420 0.0347 1.0000 1.0000
* D Col Heads
* =========
* Kg/hr Kmol/hr kmol/s wt% mol %
* Acetic Acid 0.0000 0.0000 0.0000 0.0000 0.0000
* Ethanol 423.0000 9.1957 0.0026 0.3246 0.3365
* Water 184.0000 10.2222 0.0028 0.1412 0.3741
* Butyl Acetate 1.0000 0.0086 0.0000 0.0008 0.0003
* Ethyl Acetate 695.0000 7.8977 0.0022 0.5334 0.2890
* Totals 1303.0000 27.3242 0.0076 1.0000 1.0000
*
* Total Feed State At Start of Simulation
* =====================================
* Kg/hr Kmol/hr kmol/s wt% mol %
* Acetic Acid 6424.0000 107.0667 0.0297 0.4888 0.4127
* Ethanol 5621.4000 122.2043 0.0339 0.4277 0.4710
* Water 400.6000 22.2556 0.0062 0.0305 0.0858
* Butyl Acetate 1.0000 0.0086 0.0000 0.0001 0.0000
* Ethyl Acetate 695.0000 7.8977 0.0022 0.5334 0.2890
* Totals 13142.0000 259.4329 0.0721 1.0000 1.0000
*
* The next two tables show the perturbed Ethanol/Water Feed and the resulting total feed settings that give the perturbation requested
APPENDIX C. PDIST INPUT FILES

* Ethanol/Water Feed
* ===================

<table>
<thead>
<tr>
<th>Kg/hr</th>
<th>Kmol/hr</th>
<th>kmol/s</th>
<th>wt%</th>
<th>mol%</th>
</tr>
</thead>
<tbody>
<tr>
<td>Acetic Acid</td>
<td>0.0000</td>
<td>0.0000</td>
<td>0.0000</td>
<td>0.0000</td>
</tr>
<tr>
<td>Ethanol</td>
<td>4981.8000</td>
<td>108.3000</td>
<td>0.0301</td>
<td>0.9200</td>
</tr>
<tr>
<td>Water</td>
<td>433.2000</td>
<td>24.0667</td>
<td>0.0067</td>
<td>0.0800</td>
</tr>
<tr>
<td>Butyl Acetate</td>
<td>0.0000</td>
<td>0.0000</td>
<td>0.0000</td>
<td>0.0000</td>
</tr>
<tr>
<td>Ethyl Acetate</td>
<td>0.0000</td>
<td>0.0000</td>
<td>0.0000</td>
<td>0.0000</td>
</tr>
</tbody>
</table>

** Totals ** 5415.0000 132.3667 0.0368 1.0000 1.0000

* Total Feed State After Perturbation
* -----------------------------------

<table>
<thead>
<tr>
<th>Kg/hr</th>
<th>Kmol/hr</th>
<th>kmol/s</th>
<th>wt%</th>
<th>mol%</th>
</tr>
</thead>
<tbody>
<tr>
<td>Acetic Acid</td>
<td>6424.0000</td>
<td>107.0667</td>
<td>0.0297</td>
<td>0.4888</td>
</tr>
<tr>
<td>Ethanol</td>
<td>5404.8000</td>
<td>117.4957</td>
<td>0.0326</td>
<td>0.4113</td>
</tr>
<tr>
<td>Water</td>
<td>617.2000</td>
<td>34.2889</td>
<td>0.0095</td>
<td>0.0470</td>
</tr>
<tr>
<td>Butyl Acetate</td>
<td>1.0000</td>
<td>0.0086</td>
<td>0.0000</td>
<td>0.0001</td>
</tr>
<tr>
<td>Ethyl Acetate</td>
<td>695.0000</td>
<td>7.8977</td>
<td>0.0022</td>
<td>0.0529</td>
</tr>
</tbody>
</table>

** Totals ** 13142.0000 266.7576 0.0741 1.0000 1.0000

* The following graph outlines the perturbations taking place in the Ethanol/Water Feed.
* Here time 0 represents the steady state point.
* ~
* 0.96 wt% |-----|------------------|
| | * |
| 0.92 wt% | *-------------------|
| |------------------------|
| 0.0 0.5 3.5 4.0 5.0 6.0 7.0 Time hrs
number_of_comps=5
number_of_stages=24
number_of_feeds=1
}
COLUMN_SIZE_END

COMPONENT_NAMES_BEGIN
{
  component1="acetic acid"
  component2="ethanol"
  component3="water"
  component4="butyl acetate"
  component5="ethyl acetate"
}
COMPONENT_NAMES_END

FEED_DATA_BEGIN
{
  feedstage1 stage24
  {
    flowrate=0.00386 /* flowrate (kmols/sec) */
    xmolfrac1=0.0 /* mole fractions */
    xmolfrac2=0.0
    xmolfrac3=0.0
    xmolfrac4=0.48
    xmolfrac5=0.52
    q_value=1.0
    temperature=300.0 /* not used */
    pressure=1.0 /* not used */
  }
}
FEED_DATA_END

THERMO_DATA_BEGIN
{
  MOLECULAR_WEIGHTS
  {
    component1=60.052
    component2=46.069
    component3=18.015
    component4=116.16
    component5=88.107
  }
  DENSITIES /* all densities are in (kg/m^3) */
  {
    component1=1049.0
    component2=789.0
  }
}
component3=1000.0  
component4=898.0  
component5=901.0  
}  

/* All antoine coefficients refer to the following equation: */  
/* ln(p*) = A-B/(T+C) where p* = mmHg and T in Kelvin. */  

ANTOINE_A  
{  
    component1=16.8080  
    component2=18.9119  
    component3=18.3036  
    component4=16.1836  
    component5=16.1516  
}  

ANTOINE_B  
{  
    component1=3405.57  
    component2=3803.98  
    component3=3816.44  
    component4=3151.09  
    component5=2790.5  
}  

ANTOINE_C  
{  
    component1=-56.34  
    component2=-41.68  
    component3=-46.13  
    component4=-66.15  
    component5=-57.15  
}  

THERMO_DATA_END  

SIMULATION_DATA_BEGIN  
{  
    finish_time=25000.0 /* Total period for simulation (secs) */  
    time_step=0.1 /* integration time step (secs) */  
    hist_step=25.0 /* history save time step (secs) */  
    control_plate=12 /* should be set to approx nplates/2 */  
}  

SIMULATION_DATA_END  

COLUMN_DATA_BEGIN  
{
plate_efficiency=0.65 /* murphree efficiency for plates */
total_pressure=1163.0 /* total pressure in mmHg. */

COLUMN_DATA_END

HYDRAULIC_DATA_BEGIN
{
  internal_diameter=0.0 /* (m) */
  active_plate_area=0.366 /* (m^2) */
  real_plate_area=0.455 /* (m^2) */
  downcomer_area=0.0 /* (m^2) */
  min_downcomer_area=0.0 /* (m^2) */
  area_of_holes=0.0 /* (m^2) */
  diameter_of_holes=0.0 /* (m) */
  number_of_holes=0.0
  apron_height=0.0 /* (m) */
  weir_height=0.0381 /* (m) */
  weir_length=0.5319 /* (m) */
  orifice_coefficient=0.0
  aeration_factor=0.6
}
HYDRAULIC_DATA_END

CONTROL_DATA_BEGIN
{
  REFLUX_RATE default 4
  {
    controller1
    {
      description "Fixed Reflux Rate"
      type "Fixed rate controller"
      parameter1 "reflux rate (kmols/sec)" 0.0474
    }
    controller2
    {
      description "Rate controlled by tops product composition"
      type "Proportional/integral controller"
      parameter1 "setpoint" 0.99
      parameter2 "controller gain" 4.0
      parameter3 "controller reset" 40.0
      parameter4 "mvc component" 0.0
    }
    controller3
    {
      description "liquid/liquid split"
      type "NRTL for lle with No Dynamics"
    }
  }

APPENDIX C. PDIST INPUT FILES
APPENDIX C. PDIST INPUT FILES

parameter1 "reflux ratio of ester phase" 1.42
parameter2 "component rich layer to find" 5.0
/* for rich phase +ve means reflux, -ve means take as tops */
parameter3 "temperature (K)" 303.0
parameter4 "pressure (bar)" 1.34
parameter5 "ppds vle method" 8.0 /* unifac */
parameter6 "reflux layer holdup (m⁻³)" 15.7
parameter7 "tops layer holdup (m⁻³)" 1.0
parameter8 "offtake of water phase (kmols/sec)" 0.0318
/* The next parameter is a switch which tell the models */
/* to set the butyl acetate feed flowrate equal to the */
/* butyl acetate flowrate leaving the column. */
parameter9 "Butyl Acetate Recycle mode" 1.0

controller4
{
    description "liquid/liquid split"
type "NRTL for lle with Composition Dynamics"
parameter1 "reflux ratio of ester phase" 1.42
parameter2 "component rich layer to find" 5.0
/* for rich phase +ve means reflux, -ve means take as tops */
parameter3 "temperature (K)" 303.0
parameter4 "pressure (bar)" 1.34
parameter5 "ppds vle method" 8.0 /* unifac */
parameter6 "reflux layer holdup (m⁻³)" 15.7
parameter7 "tops layer holdup (m⁻³)" 1.0
parameter8 "offtake of water phase (kmols/sec)" 0.0318
/* The next parameter is a switch which tell the models */
/* to set the butyl acetate feed flowrate equal to the */
/* butyl acetate flowrate leaving the column. */
parameter9 "Butyl Acetate Recycle mode" 1.0
}

TOPS_PRODUCT default 3
{
    controller1
    {
        description "Tops rate controlled by level in tank"
type "Prop/Integral level controller"
parameter1 "reflux drum holdup (kmols)" 5.0
parameter2 "setpoint (kmols)" 2.5
parameter3 "controller gain" 4.0
parameter4 "controller reset" 40.0
    }
    controller2
    {
        description "Tops rate controlled by level in tank"
APPENDIX C. PDIST INPUT FILES

```plaintext
type "Proportional level controller"
parameter1 "reflux drum holdup (kmols)" 5.0
parameter2 "setpoint (kmols)" 2.5
parameter3 "controller gain" 4.0
}
controller3
{
  description "no controller. flow controlled by lle split"
  type "null controller"
}

BOILUP_RATE default 4
{
  controller1
  {
    description "Fixed Boilup Rate"
    type "Fixed rate controller"
    parameter1 "reboil rate (kmols/sec)" 0.082
  }
  controller2
  {
    description "Rate controlled by bottoms product composition"
    type "Proportional/integral controller"
    parameter1 "setpoint" 0.99
    parameter2 "controller gain" 4.0
    parameter3 "controller reset" 40.0
  }
  controller3
  {
    description "Reactor/reboiler"
    type "Explicit Integration Model"
    parameter1 "feed rate (kmols/s)" 0.0721
    parameter2 "acetic acid feed conc" 0.4127
    parameter3 "ethanol feed conc" 0.4710
    parameter4 "water feed conc" 0.0858
    parameter5 "butyl acetate conc" 0.0
    parameter6 "ethyl acetate conc" 0.0305
    parameter7 "reaction pressure (bar)" 1.55
    parameter8 "reactor initial holdup(m^3)" 12.0
  }

  /* The next parameter allows the user to select a different */
  /* holdup for the reaction compared to the dynamic holdup. */
  /* This is only supposed to be used when a steady state is */
  /* being searched for. By reducing the dynamic holdup the */
  /* simulation reacts quicker and thus takes less time. The */
  /* reactor holdup must be set at the actual holdup for the */
  /* desired column since the reaction rate varies with holdup */
```
APPENDIX C. PDIST INPUT FILES

parameter9 "reaction holdup correction(m^-3)" 12.0
parameter10 "catalyst concentration" 2.5

/* the next parameters control the ramping and oscillation */
/* of the feed into the reactor reboiler. They work in the */
/* same way as the standard feed changes. See USER_DATA. */

parameter11 "set ramping mode" 0.0 /* 0 for off 1 for on */
parameter12 "ramping period" 0.0 /* time for ramp */
parameter13 "set oscillation mode" 0.0 /* 0 for off 1 for on */
parameter14 "oscillation period" 0.0 /* oscillation time */

controller4
{

description "Reactor/reboiler"
type "Implicit Integration Model"
parameter1 "feed rate (kmols/s)" 0.0721
parameter2 "acetic acid feed conc" 0.4127
parameter3 "ethanol feed conc" 0.4710
parameter4 "water feed conc" 0.0858
parameter5 "butyl acetate conc" 0.0
parameter6 "ethyl acetate conc" 0.0305
parameter7 "reaction pressure (bar)" 1.55
parameter8 "reactor initial holdup(m^-3)" 12.0

/* The next parameter allows the user to select a different */
/* holdup for the reaction compared to the dynamic holdup. */
/* This is only supposed to be used when a steady state is */
/* being searched for. By reducing the dynamic holdup the */
/* simulation reacts quicker and thus takes less time. The */
/* reactor holdup must be set at the actual holdup for the */
/* desired column since the reaction rate varies with holdup */

parameter9 "reaction holdup correction(m^-3)" 12.0
parameter10 "catalyst concentration" 2.5

/* the next parameters control the ramping and oscillation */
/* of the feed into the reactor reboiler. They work in the */
/* same way as the standard feed changes. See USER_DATA. */

parameter11 "set ramping mode" 0.0 /* 0 for off 1 for on */
parameter12 "ramping period" 0.0 /* time for ramp */
parameter13 "set oscillation mode" 0.0 /* 0 for off 1 for on */
parameter14 "oscillation period" 0.0 /* oscillation time */
}
APPENDIX C. PDIST INPUT FILES

BOTTOMS_PRODUCT default 3
{
  controller1
  {
    description "Bottoms rate controlled by level in tank"
    type "Prop/Integral level controller"
    parameter1 "reboiler drum holdup (kmols)" 5.0
    parameter2 "setpoint (kmols)" 2.5
    parameter3 "controller gain" 4.0
    parameter4 "controller reset" 100.0
  }
  controller2
  {
    description "Bottoms rate controlled by level in tank"
    type "Proportional level controller"
    parameter1 "reboiler drum holdup (kmols)" 5.0
    parameter2 "setpoint (kmols)" 2.5
    parameter3 "controller gain" 0.1
  }
  controller3
  {
    description "no bottoms"
    type "dummy controller"
  }
}

CONTROL_DATA_END

USER_DATA_BEGIN
{
  slot1 "VLE METHOD" 3.0 /* unifac */

  /* The following are the ppds codes. See ppds manual */
  slot2 "ppds code for acetic acid" 134.0
  slot3 "ppds code for ethanol" 93.0
  slot4 "ppds code for water" 63.0
  slot5 "ppds code for butyl acetate" 168.0
  slot6 "ppds code for ethyl acetate" 158.0

  /* The next slots are for latent heats. All in KJ/kmol */
  slot7 "latentheat for acetic acid" 23697.0
  slot8 "latentheat for ethanol" 38770.0
  slot9 "latentheat for water" 40683.0
  slot10 "latentheat for butyl acetate" 36006.0
}
APPENDIX C. PDIST INPUT FILES

slot11 "latent heat for ethyl acetate" 32238.0

/* The next two slots are the initial flowrates in the column. These can be calculated using the following formula.

\[ \text{liquid flow} = RR \times (RFEED - AQ \_OFFTAKE) + \text{MAKEUP} \]
\[ \text{vapour flow} = RR \times (RF - AQ \_OFFTAKE) + \text{MAKEUP} + \text{RFEED} \]
\[ \text{reflux rate} = \text{liquid flow} - \text{MAKEUP} \]

where: RR = Reflux Ratio, RFEED = Reactor Feed,
AQ\_OFFTAKE = aqueous offtake from liq/liq separator,
MAKEUP = Butyl Acetate Makeup Stream, RF = Reactor Feed

Performing this calculation can save a lot of time in finding the steady state.
*/

slot12 "initial liquid flow" 0.065204
slot13 "initial vapour flow" 0.140204
slot14 "initial reflux rate" 0.061344

/* The following slot allows the user to select the update time interval for vle calculations */

slot15 "vle update time" 5.0

/* The following slots are used to set and control how feed changes are handled. */

/* The next variable sets or unsets ramping. 0 for off, 1 for on */
slot16 "set ramping mode" 0.0

/* The next variable specifies the time period of the ramp */
/* If a ramp is set with 0 period it is handled as a step change */
slot17 "ramping period" 0.0

/* The next variable sets or unsets oscillation. 0 for off, 1 for on */
slot18 "set oscillation mode" 0.0

/* The next variable sets the oscillation period */
/* If an oscillation is set with 0 period it is taken as a step change */
slot19 "oscillation period" 0.0

/* The next slot switches on or off initialisation from file */
slot20 "init from file y/n" 1.0

/* The next slot switches on or off the simulation start mode */
If 0 simulation start with steady state calculations. If 1 it starts doing dynamic calculations. */

slot21 "simulation start mode" 0.0

USER_DATA_END

PROGRAM_BEGIN
{
    /* At time 100.0 the downward ramp is started. This is performed by setting the target feed condition and setting the ramp mode to 1.0 and the ramp period. */
    TIME 100.0 BOILUP_RATE controller4
    {
        parameter1 "feed rate (kmols/s)" 0.0741 /* was 0.0721 */
        parameter2 "acetic acid feed conc" 0.4014 /* was 0.4127 */
        parameter3 "ethanol feed conc" 0.4405 /* was 0.4710 */
        parameter4 "water feed conc" 0.1285 /* was 0.0858 */
        parameter5 "butyl acetate conc" 0.0 /* was 0.0 */
        parameter6 "ethyl acetate conc" 0.0296 /* was 0.0305 */
        parameter7 "reaction pressure (bar)" 1.55
        parameter8 "reactor initial holdup(m^-3)" 12.0
        /* The next parameter allows the user to select a different holdup for the reaction compared to the dynamic holdup. */
        /* This is only supposed to be used when a steady state is being searched for. By reducing the dynamic holdup the simulation reacts quicker and thus takes less time. The reactor holdup must be set at the actual holdup for the desired column since the reaction rate varies with holdup */
        parameter9 "reaction holdup correction(m^-3)" 12.0
        parameter10 "catalyst concentration" 2.5
        /* the next parameters controll the ramping and oscillation of the feed in to the reactor reboiler. They work in the same way as the standard feed changes. See USER_DATA. */
        parameter11 "set ramping mode" 1.0 /* 0 for off 1 for on */
        parameter12 "ramping period" 1800.0 /* time for ramp */
        parameter13 "set oscillation mode" 0.0 /* 0 for off 1 for on */
        parameter14 "oscillation period" 0.0 /* oscillation time */
    }
    /* The simulation has now run for 3.5 hours since the start of the down ramp. This is 0.5 hours for the ramp and 3 hours of simulation at the new feed condition. The next event ramps */
/* the feed back to its initial condition. Again this is set to */
/* 0.5 hours for the ramp. */

TIME 12700.0 BOILUP_RATE controller4
{
  parameter1 "feed rate (kmols/s)" 0.0721
  parameter2 "acetic acid feed conc" 0.4127
  parameter3 "ethanol feed conc" 0.4710
  parameter4 "water feed conc" 0.0858
  parameter5 "butyl acetate conc" 0.0
  parameter6 "ethyl acetate conc" 0.0305
  parameter7 "reaction pressure (bar)" 1.55
  parameter8 "reactor initial holdup(m^3)" 12.0

  /* The next parameter allows the user to select a different */
  /* holdup for the reaction compared to the dynamic holdup. */
  /* This is only supposed to be used when a steady state is */
  /* being searched for. By reducing the dynamic holdup the */
  /* simulation reacts quicker and thus takes less time. The */
  /* reactor holdup must be set at the actual holdup for the */
  /* desired column since the reaction rate varies with holdup */
  parameter9 "reaction holdup correction(m^3)" 12.0
  parameter10 "catalyst concentration" 2.5

  /* the next parameters control the ramping and oscillation */
  /* of the feed in to the reactor reboiler. They work in the */
  /* same way as the standard feed changes. See USER_DATA. */
  parameter11 "set ramping mode" 1.0 /* 0 for off 1 for on */
  parameter12 "ramping period" 1800.0 /* time for ramp */
  parameter13 "set oscillation mode" 0.0 /* 0 for off 1 for on */
  parameter14 "oscillation period" 0.0 /* oscillation time */
}

PROGRAM_END

END

C.3 PDist Graphical Output Description File

This section contains an example PDist graphical output description.
begin

# Define the reflux solution vectors

begin reflux
  slot 1 for \$\{NCOMP\} "Molefraction \$\{COMP\}" "Liq Mfrac \$\{COMP\}"
  slot (\$\{NCOMP\}+1) for \$\{NCOMP\} "Molefraction \$\{COMP\}" "Vap Mfrac \$\{COMP\}"
  slot (2*\$\{NCOMP\}+1) "Total Molar Liquid Flow" "Reflux (Kmols/s)"
  slot (2*\$\{NCOMP\}+2) "Total Molar Vapour Flow" "Tops Prod. (Kmols/s)"
  slot (2*\$\{NCOMP\}+3) "Temperature (K)" "Temperature (K)"
end

# Define the stage solution vectors

begin stage
  slot 1 for \$\{NCOMP\} "Molefraction \$\{COMP\}" "Liq. Mfrac \$\{COMP\}"
  slot (\$\{NCOMP\}+1) for \$\{NCOMP\} "Molefraction \$\{COMP\}" "Vap. Mfrac \$\{COMP\}"
  slot (2*\$\{NCOMP\}+1) "Total Molar Liquid Flow" "Liq. Flow (Kmols/s)"
  slot (2*\$\{NCOMP\}+2) "Total Molar Vapour Flow" "Vap. Flow (Kmols/s)"
  slot (2*\$\{NCOMP\}+3) "Temperature (K)" "Temperature (K)"
  slot (2*\$\{NCOMP\}+4) for \$\{NCOMP\} "RV of \$\{COMP\}" "RV \$\{COMP\}"
end

# Define the reboiler solution vectors

begin reboiler
  slot 1 for \$\{NCOMP\} "Molefraction \$\{COMP\}" "Liq. Mfrac \$\{COMP\}"
  slot (\$\{NCOMP\}+1) for \$\{NCOMP\} "Molefraction \$\{COMP\}" "Vap. Mfrac \$\{COMP\}"
  slot (2*\$\{NCOMP\}+1) "Total Molar Liquid Flow" "Bottoms (Kmols/s)"
  slot (2*\$\{NCOMP\}+2) "Total Molar Vapour Flow" "Boilup (Kmols/s)"
  slot (2*\$\{NCOMP\}+3) "Temperature (K)" "Temperature (K)"
  slot (2*\$\{NCOMP\}+4) for \$\{NCOMP\} "RV of \$\{COMP\}" "RV \$\{COMP\}"
end

# calculate the molecular weight sum for mole to mass fraction conversion

MWTSUM=
do i=1 for \$\{NCOMP\}
  MWTSUM=( \$\{MWTSUM\} + (\$\{SLOT\{i\}\} * \$\{MWT\{i\}\}))
done

# Describe 3D graphing recommendations

begin 3d
  context "Flowrates" [:]
    stage menu "Flowrates"
item 1 "Total Liquid Flowrate" slot (2*$\{\text{NCOMP}\}$+1)
item 2 "Total Vapour Flowrate" slot (2*$\{\text{NCOMP}\}$+2)

context "Liq Molefractions" [0.0:1.0]
  stage menu "Liq Molefractions"
  item 1 for $\{\text{NCOMP}\}$ "$\{\text{COMP}\}$" slot $\{\text{INDEX}\}$
  item ($\{\text{NCOMP}\}$+1) for $\{\text{NCOMP}\}$ "wt frac $\{\text{COMP}\}$"
    eval ( ($\{\text{MWTS}\{\text{BASEINDEX}}\} \times $\{\text{SLOT}\{\text{BASEINDEX}}\}$) / $\{\text{MWTSUM}}\}$)
  "wt frac $\{\text{COMP}\}$"

context "Vap Molefractions" [0.0:1.0]
  stage menu "Vap Molefractions"
  item 1 for $\{\text{NCOMP}\}$ "$\{\text{COMP}\}$" slot ($\{\text{INDEX}\}$+$\{\text{NCOMP}\}$)

context "Thermodynamics" [:] 
  stage menu "Thermodynamics"
  item 1 for $\{\text{NCOMP}\}$ "RV of $\{\text{COMP}\}$" slot (2*$\{\text{NCOMP}\}$+$3+$$\{\text{BASEINDEX}}\}$)
  item ($\{\text{NCOMP}\}$+1) "Temperature (K)" slot (2*$\{\text{NCOMP}\}$+$3$)
  item ($\{\text{NCOMP}\}$+2) "Temperature (\degree C)" eval ($\{$\text{SLOT}(2*$\{\text{NCOMP}\}$+$3$)\}' - 273.0)
    "Temperature (\degree C)"
end

# Describe 2D graphing recommendations

begin 2d
context "Flowrates" [::
  reflux menu "Flowrates"
    item 1 "Reflux Rate" slot (2*$\{\text{NCOMP}\}$+1)
    item 2 "Tops Product Rate" slot (2*$\{\text{NCOMP}\}$+2)

  stage menu "Flowrates"
    item 1 "Total Liquid Flow" slot (2*$\{\text{NCOMP}\}$+1)
    item 2 "Total Vapour Flow" slot (2*$\{\text{NCOMP}\}$+2)
    item 3 for $\{\text{NCOMP}\}$ "Liquid flowrate of $\{\text{COMP}\}$"
      eval ($\{$\text{SLOT}(2*$\{\text{NCOMP}\}$+1$)\} \times $\{\text{SLOT}\{\text{BASEINDEX}}\}'\}$)
    "Liq. Flow. $\{\text{COMP}\}$"
    item (3+$\{\text{NCOMP}\}$) for $\{\text{NCOMP}\}$ "Vapour flowrate of $\{\text{COMP}\}$"
      eval ($\{$\text{SLOT}(2*$\{\text{NCOMP}\}$+2$)\} \times $\{\text{SLOT}\{\text{BASEINDEX}}\}+$$\{\text{NCOMP}}\}'\}$)
    "Vap. Flow. $\{\text{COMP}\}$"
  reboiler menu "Flowrates"
    item 1 "Bottoms Product" slot (2*$\{\text{NCOMP}\}$+1)
    item 2 "Reboil Rate" slot (2*$\{\text{NCOMP}\}$+2)

context "Compositions" [0.0:1.0]
  reflux menu "Compositions"
    item 1 for $\{\text{NCOMP}\}$ "Liquid $\{\text{COMP}\}$" slot $\{\text{INDEX}\}$
stage menu "Compositions"
item 1 for ${NCOMP} "Liquid ${COMP}" slot ${INDEX}
item ({$NCOMP}+1) for ${NCOMP} "Vapour ${COMP}" slot ${INDEX}

reboiler menu "Compositions"
item 1 for ${NCOMP} "Liquid ${COMP}" slot ${INDEX}
item ({$NCOMP}+1) for ${NCOMP} "Vapour ${COMP}" slot ${INDEX}

context "Thermodynamics"[:]
reflux menu "Thermodynamics"
item 1 "Temperature (K)" slot (2*${NCOMP}+3)

stage menu "Thermodynamics"
item 1 for ${NCOMP} "Relative Volatility ${COMP}" slot (2*{$NCOMP}+3+{$BASEINDEX})
item ({$NCOMP}+1) "Temperature (K)" slot (2*{$NCOMP}+3)

reboiler menu "Thermodynamics"
item 1 for ${NCOMP} "Relative Volatility ${COMP}" slot (2*{$NCOMP}+3+{$BASEINDEX})
item ({$NCOMP}+1) "Temperature (K)" slot (2*{$NCOMP}+3)
end

# Describe the tabling graphing recommendations

begin tables
context "Temperature"
reflux menu "Temperature"
item 1 "Temp (K)" slot (2*${NCOMP}+3)
stage menu "Temperature"
item 1 "Temp (K)" slot (2*{$NCOMP}+3)
reboiler menu "Temperature"
item 1 "Temp (K)" slot (2*{$NCOMP}+3)

context "Flows"
reflux menu "Flows"
item 1 "Reflux" slot (2*{$NCOMP}+1)
item 2 "Tops" slot (2*{$NCOMP}+2)
item 3 "Ref Ratio"
eval (${SLOT'}(2*{$NCOMP}+1)'}/${SLOT'}(2*{$NCOMP}+2)') "Ref Ratio"

stage menu "Flows"
item 1 "Liquid" slot (2*{$NCOMP}+1)
item 2 "Vapour" slot (2*{$NCOMP}+2)
reboiler menu "Flows"
item 1 "Bottoms" slot (2*{$NCOMP}+1)
item 2 "Boilup" slot (2*{$NCOMP}+2)
item 3 "B'up Ratio"
APPENDIX C. PDIST INPUT FILES

```
eval ({$(SLOT'(2*$NCOMP)+2)'/$(SLOT'(2*$NCOMP)+1)}) "B'up Ratio"

context "Compositions"
  reflux menu "Tops Compositions"
    item 1 for $NCOMP "$COMP" slot $BASEINDEX
  stage menu "Compositions"
    item 1 for $NCOMP "$COMP" slot $INDEX
  reboiler menu "Bottoms Compositions"
    item 1 for $NCOMP "$COMP" slot $INDEX

context "Compositions"
  reflux menu "Tops Mass Compositions"
    item 1 for $NCOMP "$COMP"
      eval ({$MWT$BASEINDEX}*{$SLOT$BASEINDEX}) / {$MWT$SUM}) 
      "wt frac $COMP"
  stage menu "Mass Compositions"
    item 1 for $NCOMP "$COMP"
      eval ({$MWT$BASEINDEX}*{$SLOT$BASEINDEX}) / {$MWT$SUM}) 
      "wt frac $COMP"
  reboiler menu "Bottoms Mass Compositions"
    item 1 for $NCOMP "$COMP"
      eval ({$MWT$BASEINDEX}*{$SLOT$BASEINDEX}) / {$MWT$SUM}) 
      "wt frac $COMP"

end

# Describe the runtime graphing recommendations

begin runtime
  slot 1 for $NCOMP [0.0:1.0]
end

end
```
Appendix D

PNet Example Input Files

This appendix contains all of the example PNet input files used for the various PNet tests described in Chapter 6. The testing described was carried out in two parts. The first of these focussed on using a PNet compliant version of the steady state simulator ESSPROS. The second set of tests moved on to examine PNets performance on actual dynamic simulations. The two sets of respective input files are described separately.

D.1 Esspross Examples

This section contains the input files used for the ESSPROS testing of PNet. In total two ESSPROS examples were described. For each example, the original ESSPROS program is shown along with the created PNet input description.

D.1.1 ESSPROS Example Program 1

The first ESSPROS example is of a simple reactor and separator with recycle. The program used is as follows:
To run this program under PNet, the program description is converted to a PNet input description, where the fortran routines are replaced by programs.

D.1.2 PNet Input for ESSROSS Example Program 1

The PNet input description for ESSPROS example 1 is as follows:

```
begin "ess2pnet converted file"
#
begin pipe_format 3
    slot 1 "comp1"
    slot 2 "comp2"
    slot 3 "comp3"
end pipe_format

setstream "1" [100,100,0]
```
# mixer unit
begin process "unit1"
inpipes ["1","2"]
outpipes ["3"]
program "mixer program"
  process 0
    exec "mixer"
    imports ["int nc","3"]
    proc_type "t8"
end process

# reactor unit
begin process "unit2"
inpipes ["3"]
outpipes ["4"]
program "reactor program"
  process 0
    exec "reactor"
    imports ["int nc","3",
              "string params","-1,-1,1;",
              "doublevec realvec","1 0.9"]
    proc_type "t8"
end process

# separator unit
begin process "unit3"
inpipes ["4"]
outpipes ["5","6"]
program "separator program"
  process 0
    exec "separator"
    imports ["int nc","3",
              "string params","1,1,1;",
              "doublevec realvec","1"]
    proc_type "t8"
end process

# splitter unit
begin process "unit4"
inpipes ["5"]
outpipes ["2","7"]
program "splitter program"
  process 0
    exec "splitter"
    imports ["int nc","3",
              "doublevec realvec","0.75"]
D.1.3 ESSPROS Example Program 2

The second ESSPROS example is much more complex. This time the process simulated represents an oil rig separation system. The example is taken from a Final Year project performed for Shell UK.

! F8X Fortran program generated by Esspros Tool
! implicit integer (a-z)
call setnc (19)
call feed(1,'0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0;')
call feed(2,'310.5,166.5,134.3,19.9,77.4,31.5,57.7,72.5,47.9,105.9,142.9,
154.5,138.0,100.9,58.4,34.9,11.8,22.7,0.06;')
call recycle(3,'0;')
call recycle(4,'0;')
call recycle(5,'0;')
do
call mixer(1,2,6)
call flash(6,7,8,param('79.73,26.62,11.58,6.09,4.96,2.62,2.21,1,.448,.0858,
.0214,.0054,.0003,0,0,0,149.5,37.18,28.79;'),8,0.0491).
call separator(7,9,10,param('.95,.95,.95,.95,.95,.95,.95,.95,.95,.95,.95,.95,.95,.95,.95,.95,.95,.95,.95,.95,.95,.95,.95,.95,.95,.95,.95,.95,.95,.95,.95,.95,.95;.1),1)
call mixer(3,9,11)
call flash(11,12,13,param('201.9,51.14,18.5,8.89,6.75,3.23,2.45,1.0,.409,
.057,.014,0,0,0,0,0,1820,123,45;'),8,0.1713)
call separator(8,14,15,param('1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1;'),1)
call mixer(13,14,16)
call flash(16,17,18,param('174,48.16,18.0,8.62,6.66,3.13,2.56,1.0,.381,.0515,
.0103,.0012,.0001,0,0,0,383,78.3,30.18;'),8,0.0519)
call mixer(4,17,19)
call flash(19,3,20,param('536,104.9,31.35,13.09,9.42,3.93,3.0,1.0,.311,.0324,
.0067,0,0,0,0,0,1625,235,60.8;'),8,0.138)
To run this program under PNet, the program description is converted to a PNet input description, were the fortran routines are replaced by programs.

D.1.4 PNet Input for ESSROSS Example Program 2

The PNet input description for ESSPROS example 2 is as follows:

```
begin "ess2prun converted file"

# output format
begin pipe_format 19
  slot 1 "comp1"
  slot 2 "comp2"
  slot 3 "comp3"
  slot 4 "comp4"
  slot 5 "comp5"
  slot 6 "comp6"
  slot 7 "comp7"
  slot 8 "comp8"
  slot 9 "comp9"
  slot 10 "comp10"
  slot 11 "comp11"
  slot 12 "comp12"
  slot 13 "comp13"
  slot 14 "comp14"
  slot 15 "comp15"
  slot 16 "comp16"
  slot 17 "comp17"
```
APPENDIX D. PNET EXAMPLE INPUT FILES

slot 18 "comp18"
slot 19 "comp19"
end pipe_format

setstream "1" [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]
setstream "2" [310.5,166.5,134.3,19.9,77.4,31.5,57.7,72.5,47.9,
105.9,142.9,154.5,138.0,100.9,58.4,34.9,11.8,22.7,0.06]

# mixer unit
begin process "unit1"
  inpipes ["1","2"]
  outpipes ["6"]
  program "mixer program"
  process 0
    exec "mixer"
    imports ["int nc","19"]
    proc_type "t8"
end process

# flash unit
begin process "unit2"
  inpipes ["6"]
  outpipes ["7","8"]
  program "flash program"
  process 0
    exec "flash"
    imports ["int nc","19",
      "string params","79.73,26.62,11.58,6.09,4.96,2.62,2.21,1,.448,
      .0858,.0214,.0054,.0003,0,0,0,149.5,37.18,28.79;",
      "doublevec realvec","8 0.0491"]
    proc_type "t8"
end process

# separator unit
begin process "unit3"
  inpipes ["7"]
  outpipes ["9","10"]
  program "separator program"
  process 0
    exec "separator"
    imports ["int nc","19",
      "string params",".95,.95,.95,.95,.95,.95,.95,.95,.95,.95,.95,.95,.95,.95,.95,.95,.95,.95,.95,.95,.95,.95,.95,.95,.95,.95;",
      "doublevec realvec","1"]
    proc_type "t8"
end process
# mixer unit
begin process "unit4"
inpipes ["3","9"]
outpipes ["11"]
program "mixer program"
  process 0
    exec "mixer"
    imports ["int nc","19"]
    proc_type "t8"
end process

# flash unit
begin process "unit5"
inpipes ["11"]
outpipes ["12","13"]
program "flash program"
  process 0
    exec "flash"
    imports ["int nc","19",
            "string params","201.9,51.14,18.5,8.89,6.75,3.23,2.45,1.0,.409,.057,.014,0,0,0,0,1820,123,45;",
            "doublevec realvec","8 0.1713"]
    proc_type "t8"
end process

# separator unit
begin process "unit6"
inpipes ["8"]
outpipes ["14","15"]
program "separator program"
  process 0
    exec "separator"
    imports ["int nc","19",
             "string params","1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1;",
             "doublevec realvec","1"]
    proc_type "t8"
end process

# mixer unit
begin process "unit7"
inpipes ["13","14"]
outpipes ["16"]
program "mixer program"
  process 0
    exec "mixer"
APPENDIX D. PNET EXAMPLE INPUT FILES

imports ["int nc","19"]
proc_type "t8"
end process

# flash unit
begin process "unit8"
inpipes ["16"]
outpipes ["17","18"]
program "flash program"
process 0
exec "flash"
imports ["int nc","19",
"string params","174,48.16,18.0,8.62,6.66,3.13,2.56,1.0,.381,.0515,.0103,.0012,.0001,0,0,0,383,78.3,30.18;",
"doublevec realvec","8 0.0519"]
proc_type "t8"
end process

# mixer unit
begin process "unit9"
inpipes ["4","17"]
outpipes ["19"]
program "mixer program"
process 0
exec "mixer"
imports ["int nc","19"]
proc_type "t8"
end process

# flash unit
begin process "unit10"
inpipes ["19"]
outpipes ["3","20"]
program "flash program"
process 0
exec "flash"
imports ["int nc","19",
"string params","536,104.9,31.35,13.09,9.42,3.93,3.0,1.0,.311,.0324,.0067,0,0,0,0,0,0,1625,235,60.8;",
"doublevec realvec","8 0.138"]
proc_type "t8"
end process

# mixer unit
begin process "unit11"
APPENDIX D. PNET EXAMPLE INPUT FILES

```plaintext
inpipes ["5","18"]
outpipes ["21"]
program "mixer program"
    process 0
        exec "mixer"
        imports ["int nc","19"]
        proc_type "t8"
    end process

# mixer unit
begin process "unit12"
inpipes ["20","21"]
outpipes ["22"]
program "mixer program"
    process 0
        exec "mixer"
        imports ["int nc","19"]
        proc_type "t8"
end process

# flash unit
begin process "unit13"
inpipes ["22"]
outpipes ["23","24"]
program "flash program"
    process 0
        exec "flash"
        imports ["int nc","19",
            "string params","230,61.55,20.93,9.38,7.34,3.32,2.69,1.0,..369,
            .0466,.0082,.0009,0,0,0,0,497.5,86.8,29.56;",
            "doublevec realvec","8 0.154"]
        proc_type "t8"
end process

# flash unit
begin process "unit14"
inpipes ["23"]
outpipes ["4","5"]
program "flash program"
    process 0
        exec "flash"
        imports ["int nc","19",
            "string params","423,88.1,28.63,13.0,8.82,3.79,2.89,1.0,..328,
            .0352,.0068,.0006,0,0,0,0,950,129.4,30;",
            "doublevec realvec","8 0.9867"]
        proc_type "t8"
```
The dynamic simulation tests were performed using connected PDists. The two examples run were very similar. The first was of a simple linear train of three columns. The second was of the same train with a recycle. The recycle being put in to full test the simulators independent connection strategy.

The actual input descriptions are small. This was made possible by using the system command, described in section 5.4.3, to get PDist to create the simulator loading descriptions required. The two example input files are shown below. To highlight the simplification which the system call mechanism provides, the created load map for a single PDist is shown at the end of the section.

D.2.1 Example 1: PDist Linear Train Simulation

This is the PNet input description for the simple linear column train using PDist.

```
# PNet Example 1:
# Simple Linear Train of 3 Distillation Columns.
# Separation is of Methanol, Ethanol. Propanol and Butanol.
#
begin "Alcohol Separation Sequence"

# Describe the pipe stream data format.
```
### APPENDIX D. PNÉT EXAMPLE INPUT FILES

```
begin pipe-format 7
  slot 1 "Flowrate"
  slot 2 "mf Methanol"
  slot 3 "mf Ethanol"
  slot 4 "mf Propanol"
  slot 5 "mf Butanol"
  slot 6 "Temp (K)"
  slot 7 "Q value"
end pipe-format

# # Column 1
#

begin process "Column 1"
  inpipes ["1"]
  outpipes ["2","3"]
  # Use system call on PDist for program parameters
  system pdist -p 5 -config MKCS1a DEFAULT -group 0 -d -map PNET mepb1.dat
end process

begin process "Column 2"
  inpipes ["3"]
  outpipes ["4","5"]
  system pdist -p 5 -config MKCS1a DEFAULT -group 1 -d -map PNET mepb2.dat
end process

begin process "unit 3"
  inpipes ["5"]
  outpipes ["6","7"]
  system pdist -p 5 -config MKCS1a DEFAULT -group 2 -d -map PNET mepb3.dat
end process

# Set the feed stream to column 1
setstream "1" [0.015, 0.3, 0.3, 0.2, 0.2, 300.0, 1.0]

end
```
D.2.2 Example 2: PDist Train Simulation with Recycle

This is the PNet input description for the PDist column train with recycle. The description includes loading instructions for a mixer and splitter program.

#
# PNet Example 1:
# Train of 3 Distillation Columns with Recycle
# Separation is of Methanol, Ethanol, Propanol and Butanol.
#
begin "Alcohol Separation Sequence"

# Describe the pipe stream data format.

begin pipe_format 7
    slot 1 "Flowrate"
    slot 2 "mf Methanol"
    slot 3 "mf Ethanol"
    slot 4 "mf Propanol"
    slot 5 "mf Butanol"
    slot 6 "Temp (K)"
    slot 7 "Q value"
end pipe_format

#
# Column 1
#

begin process "Recycle Mixer"
    inpipes ["1","R"]
    outpipes ["M"]
    program "Recycle Mixer"
        process 0
        exec "mixer"
        proc_type "t8"
end process

begin process "Column 1"
    inpipes ["M"]
    outpipes ["2","3"]

# Use system call on PDist for program parameters
system pdist -p 4 -config MKCS1a DEFAULT -group 0 -d -map PNET mepb1.dat
end process

begin process "Column 2"
  inpipes ["3"]
  outpipes ["4","5"]
  system pdist -p 4 -config MKCS1a DEFAULT -group 1 -d -map PNET mepb2.dat
end process

begin process "unit 3"
  inpipes ["5"]
  outpipes ["6","7"]
  system pdist -p 4 -config MKCS1a DEFAULT -group 2 -d -map PNET mepb3.dat
end process

begin process "Recycle Splitter"
  inpipes ["4"]
  outpipes ["R","8"]

  program "Dynamic Splitter"
    process 0
      exec "splitter"
      imports ["double SplitFrac","0.6"]
      proc_type "t8"
  end process

# Set the feed stream
setstream "1" [0.015, 0.3, 0.3, 0.2, 0.2, 300.0, 1.0]
end

D.2.3 Example PDist Loader Text under PNet

This final section shows the PNet loading description created for PDist by PDist itself.

begin process "Column 1"
  inpipes ["M"]
  outpipes ["2","3"]
# Use system call on PDist for program parameters
# PNet Preprocessor:
# Executed: pdist -p 4 -i -gfx ling:0.0 -config MKCS1a DEFAULT -group 0 -d -map PNET mepbl.dat | awk '{for (i=0;i<4;i++) printf(" "); print $0}'

# ================================================================================
# # PDist
#
# (A Parallel Interactive Dynamic Distillation Simulator)
# # Loader Version : 1 4/22/93
# # Developer : Roderick C. McKinnel
# # : Ecosse Project
# # Copyright : Roderick C. McKinnel, All Rights Reserved
# # ================================================================================
# # PDist: Interrogating Hardware:
# # sun4(s) available = 1
# # t8(s) available = 12
#
# PDist: Launching specific version for MKCS1_RGC hardware.
# PDist: exec = /home/rory/work/parallel/PDist/bin/pdist_MKCS1_RGC_sun4 -p 4 -
# File : PNet loading description for PDist
# Creator: PDist loader
# Author : Roderick C McKinnel

program "PDist"

process 0
  exec "pdist_master"
  args "-gfxmode -interactive mepbl.dat /home/rory/work/parallel/PDist/model/pnet_models/gfxform.dat"
  imports ["int pdistgrp","0"]
  proc_type "sun4"
  linkto [3]

process 1
  exec "pdist_interaction"
  args "mepbl.dat /home/rory/work/parallel/PDist/model/pnet_models/gfxform.dat"
  imports ["int pdistgrp","0"]
  proc_type "sun4"
process 2
exec "pdist_rgfx"
args "mepbl.dat /home/rory/work/parallel/PDist/model/pnet_models/gxf orm.dat ling:0.0"
imports ["int pdistgrp","0"]
proc_type "sun4"

process 3
exec "pdist_reb"
args ""
imports ["int pdistnProcs","4",
"int pdistprocId","1",
"int pdistlocalId","0",
"int pdistnLocals","0",
"int pdistgrp","0",
"int localbot","0",
"int nplates","0"]
proc_type "t8"
linkto [4]

process 4
exec "pdist_st"
args ""
imports ["int pdistnProcs","4",
"int pdistprocId","2",
"int pdistlocalId","0",
"int pdistnLocals","0",
"int pdistgrp","0",
"int interactive","1",
"int localbot","1",
"int nplates","20"]
proc_type "t8"
linkto [5]

process 5
exec "pdist_st"
args ""
imports ["int pdistnProcs","4",
"int pdistprocId","3",
"int pdistlocalId","0",
"int pdistnLocals","0",
"int pdistgrp","0",
"int interactive","1",
"int localbot","21",
"int nplates","20"]
proc_type "t8"
linkto [6]

process 6
exec "pdist_ref"
args ""
imports ["int pdistnProcs","4", "int pdistprocId","4", "int pdistlocalId","0", "int pdistnLocals","0", "int pdistgrp","0", "int localbot","41", "int nplates","0"]

proc_type "t8"

# end of PNet format PDist description
# PDist: Launcher completed execution.
end process
Appendix E

Syntax Diagrams for the Various Input Files Developed

This appendix contains the syntax descriptions for the input files used for PDist’s model routines, PDist’s graphical back end and PNet. The syntax for each format is presented using rail diagrams.

Rail diagrams are built out of input syntax rules. Each rule is built up from further rules and tokens. A token is a specific grouping of text within the data file. Everything which appears in a round ended box is a token. Everything in a rectangular box contains the name for a set of rules which are expanded further down the page. In the diagrams each token or rule is connected to the next one by rails. Starting from the left hand side the rail layout dictates the way in which you can pass through various tokens and internal rules. Loops in the rail layout circle rules which can be repeated before proceeding further. Branches in the rail layout provide a choice of rules to follow.

Specific text is used to describe tokens. The exception is where a choice of characters is available. A choice of one from many is shown between square brackets. e.g. \([a - z]\) means any lower case alphabetic character.
E.1 PDist Graphics Description Syntax

The following diagrams show the syntax used to describe the solutions which are exported from the PDist models.

**DataFile**

```
begin Categories end

HashC OldStyleFormat
```

The old format graphics file was based on a fixed output suite of data. The old format had a simple header followed by the solution data whose format was known. This has now been replaced by a backward compatibility format file of the type described here.

**OldStyleFormat**

```
HashC HashC Integer HashC Integer

ComponentNames HashC Real HashC Real
```

**ComponentNames**

```
HashC String DataFile
```

The contents of the format file are split into categories. The categories are either associated with solution structure declarations or for setting up variables used within these declarations.
Categories

The equality rules are relatively simple. To add functionality a simple loop structure is provided. At present this is limited to one nest deep. The reason being that loops are difficult to program in a Rule Based parser. If required this can
be added at a later date.

LoopList

\[
\text{Equality} \quad \text{ValueString} \\
\text{do} \quad \text{Equality} \quad \text{LoopRange} \quad \text{done} \\
\text{LoopList1}
\]

NB. If the "do" token in the next rule is picked up the parser fails with a "too many nests" error.

LoopList1

\[
\text{Equality} \quad \text{ValueString} \\
\text{If this track then error} \quad \text{do}
\]

ValueString

\[
\text{Integer} \\
\text{Real} \\
\text{String} \\
\text{Variable} \\
\text{Expression}
\]

DataFormat

\[
\text{slot} \quad \text{RangeType} \quad \text{String} \quad \text{String}
\]

Contexts

\[
\text{context} \quad \text{String} \quad \text{MaxAndMin} \quad \text{ContextSpecs}
\]
MaxAndMin

ContextSpecs

ContextMenu

SlotDefinition

RunTimeSlots
Variable

The following represents the format used for a mathematical expression.

Expression

Equality

Integer
APPENDIX E. INPUT FILE SYNTAX DIAGRAMS

Real

Exp

String

Digit

HashC
E.2 PDist Model Input Format

Comments may appear at any time within the format. There are two formats: Hash and enclosed comments.

Comment

DataFile

BEGIN { DataList } END
APPENDIX E. INPUT FILE SYNTAX DIAGRAMS

CSizes

\[
\begin{align*}
\text{nos-of-comps} & = \text{Integer} \\
\text{nos-of-stages} & = \text{Integer} \\
\text{nos-of-feeds} & = \text{Integer}
\end{align*}
\]

CompNms

\[
\begin{align*}
\text{Component} & = \text{String}
\end{align*}
\]

Feeds

\[
\begin{align*}
\text{feedstage} & \text{ Integer} \{ \text{FeedParameters} \}
\end{align*}
\]

FeedParameters

\[
\begin{align*}
\text{flowrate} & = \text{Real} \\
\text{xmolefrac} & \text{ Integer} = \text{Real} \\
\text{q-value} & = \text{Real} \\
\text{temperature} & = \text{Real} \\
\text{pressure} & = \text{Real}
\end{align*}
\]
ThermDyn

MOLECULAR-WEIGHTS { } ThermoData }

DENSITIES { ThermoData }

ANToine-A { ThermoData }

ANToine-B { ThermoData }

ANToine-C { ThermoData }

ThermoData

component Integer = Real

ColData

plate-efficiency = Real

total-pressure = Real

SimData

finish-time = Real

time-step = Real

hist-step = Real

control-plate = Integer
**APPENDIX E. INPUT FILE SYNTAX DIAGRAMS**

*Hydraulics*

- `internal-diameter` = `Real`
- `active-plate-area` = `Real`
- `real-plate-area` = `Real`
- `downcomer-area` = `Real`
- `min-downcomer-area` = `Real`
- `area-of-holes` = `Real`
- `diameter-of-holes` = `Real`
- `number-of-holes` = `Real`
- `apron-height` = `Real`
- `weir-height` = `Real`
- `weir-length` = `Real`
- `orifice-coefficient` = `Real`
- `aeration-factor` = `Real`

*CModes*

- `REFLUX-RATE` default `Integer` { Controllers }
- `TOPS-PRODUCT` default `Integer` { Controllers }
- `BOILUP-RATE` default `Integer` { Controllers }
- `BOTTOMS-PRODUCT` default `Integer` { Controllers }
Controllers

```
controller Integer { CSpecs }
```

CSpecs

```
description String
type String
parameter Integer String Real
```

UserData

```
slot Integer String Real
```

ProgramData

```
TIME Real ProgramEvent
STATE Integer ProgramEvent
```
ProgramEvent

- REFUX-RATE: default Integer
- TOPS-PRODUCT: default Integer
- BOILUP-RATE: default Integer
- BOTTOMS-PRODUCT: default Integer
- REFUX-RATE: controller Integer
- TOPS-PRODUCT: controller Integer
- BOILUP-RATE: controller Integer
- BOTTOMS-PRODUCT: controller Integer
- USER-DATA: slot Integer Real
- USER-DATA: slot Integer String Real
- FEED-STAGE: FeedParameters

Integer

- Digit
- [+-]
APPENDIX E. INPUT FILE SYNTAX DIAGRAMS

Real

Exp

String

Digit

[0-9]
E.3 PNet Input File Syntax

DataFile

begin String ProcessUnit end

ProcessUnit

begin process String String UnitDefs end process

begin process String UnitDefs end process

begin pipe_format Integer PFormats end pipe_format

setstream String SetStream

UnitDefs

inpipes StreamName

outpipes StreamName

pipe_imports String StrImports

program String ProgramProcesses
StreamName

SetStream

ProgramProcesses

ProcessData
**Imports**

```
String, String
String, String
```

**StrImports**

```
String, String, String
String, String
```

**LinkList**

```
Integer
Integer
Expression
Expression
```

**PFormats**

```
slot Integer String
slot Integer for Integer String
```

**Integer**

```
Digit
```

```
Real

Exp

String

Digit

[0-9]
Appendix F

Bibliography


[100] Paul Sawyer, "STEP Steps In," The Chemical Engineer (8th April, 1993).
