An Automated Parallel Application Creation and Execution Tool for Clusters

by

Christopher McAvaney

BSc (Honours)

Submitted in fulfilment of the requirements for the degree of

Doctor of Philosophy

in

Computer Science

at

Deakin University

February 2003
I certify that the thesis entitled:

*An Automated Parallel Application Creation and Execution Tool for Clusters*

submitted for the degree of:

**DOCTOR OF PHILOSOPHY**

is the result of my own work and that where reference is made to the work of others, due acknowledgement is given.

I also certify that any material in the thesis which has been accepted for a degree or diploma by any other university or institution is identified in the text.

Full Name...................................................................................................................
(Please Print)

Signature...........................................................................................................Date...............
Acknowledgements

I would like to thank my supervisor, Andrzej Goscinski. His patience, guidance and support have been invaluable throughout the period of this research.

I would like to thank my colleagues in the GENESIS research group: Justin Rough, Damien De Paoli, Michael Hobbs, Jackie Silcock and Greg Wickham for the friendship and assistance in understanding such a huge and ongoing project. Also staff members (past and present) from the School of Computing and Mathematics, Chengqi Zhang and Geoff Webb for their encouragement.

Many thanks go to my family, in particular my parents Kevin and Joy, their support and reassurance has been great help to me in continuing my studies. I would especially like to thank Robyn for her friendship, everlasting encouragement, much patience and love that she has provided me in these last few years.

I am also very grateful for the gracious support provided by Professor Pip Hamilton and Alison Hadfield within Research Services over the last year.

Finally, this work was supported by an Australian Postgraduate Awards Scholarship and the School of Computing and Mathematics, without which this research would never have been completed.
# Table of Contents

Chapter 1  Introduction ........................................................................................................... 1
  1.1 Motivation ....................................................................................................................... 1
  1.2 Research Aims ................................................................................................................ 5
  1.3 Research Methods ......................................................................................................... 6
  1.4 Overview of this Thesis ................................................................................................. 6

Chapter 2  Related Work and Concepts ............................................................................... 9
  2.1 Introduction .................................................................................................................... 9
  2.2 Parallel Application Creation ....................................................................................... 9
    2.2.1 Assisted Semi-automatic Parallelisation Techniques ........................................ 10
      Analysis .......................................................................................................................... 10
      Synchronisation ........................................................................................................... 11
      Automation .................................................................................................................... 12
  2.2.2 Survey of Assisted and Semi-automatic Parallelisation Tools ...................... 12
    2.2.2.1 ParaScope ........................................................................................................ 13
    2.2.2.2 SUIF ............................................................................................................... 13
    2.2.2.3 Polaris ............................................................................................................. 14
    2.2.2.4 Parafrase-2 .................................................................................................... 15
    2.2.2.5 Paradigm ....................................................................................................... 15
  2.2.3 Review ...................................................................................................................... 16
  2.3 Execution of Parallel Applications ............................................................................... 19
    2.3.1 Services and Characteristics Required for Automatic Parallel Processing ........ 20
    Granularity ....................................................................................................................... 20
    Parallelism Management ................................................................................................. 21
    Efficient Message Passing Mechanisms ......................................................................... 22
    Shared Memory Mechanisms ......................................................................................... 23
    Synchronisation Mechanisms ....................................................................................... 24
  2.3.2 Survey of Execution Environments ....................................................................... 25
    2.3.2.1 Beowulf .......................................................................................................... 25
    2.3.2.2 Berkeley’s NOW ............................................................................................. 27
Chapter 3 Synthesis of an Automated Parallel Application Creation and Execution Tool .......................................................................................................................... 37
  3.1 Introduction .................................................................................................................. 37
  3.2 Requirements of the tool .............................................................................................. 38
    3.2.1 Requirements of the Programmer ........................................................................ 38
    3.2.2 Automated Parallelisation Requirements ............................................................. 41
    3.2.3 Execution Environment ......................................................................................... 46
    3.2.4 Overall requirements .............................................................................................. 51
  3.3 Design of an Automated Parallel Creation and Execution Tool ....................... 51
    3.3.1 Architecture ........................................................................................................... 52
    3.3.2 Parallelising Compiler ............................................................................................ 53
      3.3.2.1 Phase 1 - Lexical, Syntax and Semantic Analysis ........................................ 55
      3.3.2.2 Phase 2 - Detection of Units of Parallelism ................................................. 56
      3.3.2.3 Phase 3 - Detection of Sequences ................................................................ 58
      3.3.2.4 Phase 4 - Insertion of Synchronisation Points ............................................. 61
      3.3.2.5 Phase 5 - Code Optimisation ......................................................................... 63
      3.3.2.6 Phase 6 - Low Level Code Generation ......................................................... 64
    3.3.3 Primitives ............................................................................................................... 65
      3.3.3.1 Primitives of the Execution Environment .................................................... 65
      3.3.3.2 Linking Primitives into the Parallel Application ............................................ 67
    3.3.4 Execution Environment .......................................................................................... 72
  3.4 Summary ...................................................................................................................... 76

Chapter 4 Implementation of the automated parallelisation creation and execution tool .......................................................................................................................... 78
  4.1 Introduction .................................................................................................................. 78
  4.2 Implementation of Phase 1 - Lexical, Syntax and Semantic Analysis .............. 79
  4.3 Implementation of Phase 2 - Detection of Units of Parallelism ..................... 82
  4.4 Implementation of Phase 3 - Detection of Sequences ....................................... 84
4.5 Implementation of Phase 4 - Insertion of Synchronisation Points .......... 89
4.6 Implementation of Phase 5 - Code Optimisation .............................. 90
4.7 Execution Environment Services Used in the Automated
Parallelisation Tool........................................................................... 91
4.7.1 Structure of GENESIS ............................................................. 92
4.7.2 Resource Discovery, Global Scheduling and
Migration Management........................................................................ 94
4.7.3 Process Creation................................................................. 94
4.7.4 Message Passing and Shared Memory....................................... 96
4.7.5 Process Synchronisation...................................................... 97
4.8 GENESIS Execution Environment Primitives .................................. 99
4.8.1 Execution Manager Primitive ............................................. 99
4.8.2 Process Manager Primitive ................................................. 102
4.8.3 DSM Manager Primitives ..................................................... 102
4.9 Implementation of Phase 6 - Low Level Code Generation .............. 105
4.9.1 Generating Parallel Applications using Message Passing .......... 106
Synchronisation and Information Dissemination.............................. 110
4.9.2 Generating Parallel Applications using Shared Memory .......... 112
Synchronisation and Information Dissemination.............................. 112
4.10 Parallel Application Execution.................................................. 115
4.11 Summary .................................................................................. 116

Chapter 5 Assessment of the Automated Parallel Application Creation and
Execution Tool .................................................................................. 119
5.1 Introduction ............................................................................. 119
5.2 Criteria for the Assessment ...................................................... 120
5.2.1 Usability Criteria.................................................................. 120
5.2.2 Performance Criteria............................................................ 121
5.3 Testing Environment ................................................................ 122
5.3.1 The Tool............................................................................. 122
5.3.2 The GENESIS Execution Environment Platform.................. 123
5.4 Objectives and Scopes of Tests ................................................ 125
5.4.1 Test programs..................................................................... 126
5.4.1.1 Towers of Hanoi ............................................................ 126
5.4.1.2 Matrix Multiplication ................................................................. 127
5.5 Results - Towers of Hanoi ................................................................. 129
  5.5.1 Qualitative Results ........................................................................ 129
    5.5.1.1 Towers of Hanoi - Message Passing based version .............. 130
    5.5.1.2 Towers of Hanoi - Shared Memory based version ............ 132
    5.5.1.3 Qualitative Summary .......................................................... 133
  5.5.2 Quantitative Results ..................................................................... 134
    5.5.2.1 Towers of Hanoi - Message Passing based version ...... 134
    5.5.2.2 Towers of Hanoi - Shared Memory based version ...... 136
  5.5.3 Towers of Hanoi Summary .............................................................. 137
5.6 Results - Matrix Multiplication ........................................................ 138
  5.6.1 Qualitative Results ........................................................................ 138
    5.6.1.1 Matrix Multiplication - Message Passing based version 138
    5.6.1.2 Matrix Multiplication - Shared Memory based version. 146
    5.6.1.3 Qualitative Summary .......................................................... 149
  5.6.2 Quantitative Results ..................................................................... 150
    5.6.2.1 Matrix Multiplication - Message Passing based version 150
    5.6.2.2 Matrix Multiplication - Shared Memory based version. 153
  5.6.3 Matrix Multiplication Summary .................................................... 154
5.7 Summary ............................................................................................. 155

Chapter 6 Conclusions and Future Work .................................................. 158
  6.1 Research Outcomes and Conclusions ............................................. 158
  6.2 Future work ....................................................................................... 164

Bibliography ............................................................................................. 167

Appendix A Example Makefile for the Matrix Multiplication
    Parallel Application ................................................................................ 179

Appendix B Example Execution Batch File ............................................. 182

Appendix C Message Passing Parent Example ..................................... 183

Appendix D Message Passing Child Example ....................................... 193
Appendix E  Message Passing Miscellaneous Generated Files .......................... 196
Appendix F  Shared Memory Parent Example ............................................. 198
Appendix G  Shared Memory Child Example .............................................. 201
Appendix H  Shared Memory Miscellaneous Generated Files ....................... 202


List of Figures

3.1  Programmer Involvement with Manual Programming Tools ........................................ 40
3.2  Phases concept ........................................................................................................... 41
3.3  Matrix multiplication example .................................................................................. 44
3.4  SPMD and MasterWorker structure .......................................................................... 45
3.5  Automated parallel creation and execution tool schema ............................................ 52
3.6  Traditional compiler architecture ............................................................................ 53
3.7  Parallelising compiler architecture .......................................................................... 55
3.8  Phase 1 information ................................................................................................ 56
3.9  Pseudo code to identify implicit blocks .................................................................... 57
3.10 Phase 2 information .................................................................................................. 58
3.11 Phase 3 information .................................................................................................. 61
3.12 Pseudo code to identify synchronisation points ....................................................... 62
3.13 Phase 4 information .................................................................................................. 63
3.14 Optimisation examples ............................................................................................. 63
3.15 Phase 5 information .................................................................................................. 64
3.16 Pseudo code for Phase 6 .......................................................................................... 71
3.17 Phase 6 information .................................................................................................. 72
3.18 Execution environment architecture ......................................................................... 73
3.19 Architecture of the whole tool ................................................................................ 77
4.1  Syntax tree structure .................................................................................................. 80
4.2  Syntax tree visualisation ........................................................................................... 81
4.3  Symbol table visualisation ......................................................................................... 82
4.4  Phase 2 pseudocode algorithm ................................................................................. 83
4.5  Parallel symbol table structure ................................................................................ 84
4.6  Parallel symbol table visualisation .......................................................................... 85
4.7  Identifier identification algorithm ............................................................................. 86
4.8  Procedure/function parameter detection algorithm ................................................... 87
4.9  Semaphore identification algorithm .......................................................................... 88
4.10 Syntax tree extract with modifications ...................................................................... 88
4.11 Identification of consecutive sequences for insertion of synchronisation points ........ 90
4.12 Code optimisation example .................................................................................. 91
4.13 GENESIS architecture .......................................................................................... 93
4.14 Process Creation Primitive .................................................................................... 100
4.15 Algorithm used to create unique parallel process name ........................................ 100
4.16 MasterWorker model implementation ..................................................................... 101
4.17 Wait primitive .......................................................................................................... 102
4.18 Shared memory creation primitive ........................................................................ 102
4.19 Shared memory lock and unlock primitives ........................................................... 104
4.20 Shared memory synchronisation primitive ............................................................ 105
4.21 Loop iteration calculation example ......................................................................... 107
4.22 Insertion of execution environment primitives pseudocode example ..................... 108
4.23 Parallel process code generation pseudocode ...................................................... 109
4.24 Algorithm for the generation of low level code to pack and send a
   message buffer. ........................................................................................................... 111
4.25 Algorithm for sending loop index information ..................................................... 111
4.26 Shared memory structure implementation ........................................................... 113
4.27 Algorithm to insert barrier() synchronisation primitive ....................................... 114
4.28 Semaphore insertion .............................................................................................. 115
4.29 Code to create execution batch script and execute the parallel application............. 116
5.1 GENESIS cluster ...................................................................................................... 124
5.2 Pseudocode of the Towers of Hanoi program ....................................................... 127
5.3 Pseudocode for Matrix Multiplication program ...................................................... 128
5.4 Towers of Hanoi source program, sequential version ........................................... 129
5.5 Parallel symbol table for Tower of Hanoi program ................................................ 130
5.6 Parallel version of the Towers of Hanoi test program using message passing 131
5.7 Parallel version of the Towers of Hanoi test program using shared memory.. 133
5.8 Time stamp locations ............................................................................................ 134
5.9 Towers of Hanoi with 20discs - Message Passing ................................................ 135
5.10 Towers of Hanoi with 20discs - Shared Memory .................................................. 137
5.11 Matrix Multiplication source program, sequential version .................................. 139
5.12 Parallel symbol table for Matrix Multiplication program .................................... 140
5.13 Non-parallelisation of initialisation loops ............................................................. 142
5.14 Dependencies on the sum variable ....................................................................... 142
5.15 Parallel version of the Matrix Multiplication test program using

   message passing................................................................. 143

5.16 Code segment for sending information to child process.......................... 145

5.17 Parallel version of the Matrix Multiplication test program using

   message passing................................................................. 145

5.18 Parallel version of the Matrix Multiplication test program using

   shared memory................................................................. 147

5.19 Matrix Multiplication - Message Passing (single parallel section)........... 150

5.20 Matrix Multiplication - Message Passing (multiple parallel sections)........ 152

5.21 Matrix Multiplication - Shared Memory ...................................... 153
List of Tables

3.1 Table of proposed primitives and their semantics ........................................... 67
5.1 Towers of Hanoi, coefficient of determination values for
   Message Passing and Shared Memory tests...................................................... 135
5.2 Matrix Multiplication, coefficient of determination values for
   Message Passing and Shared Memory tests...................................................... 151
Abstract

Current trends in parallel processing are utilising clusters to achieve solutions quickly, economically and efficiently. The cost and usability of super-computers is prohibitive for many. However there are two issues that hamper parallel processing on clusters mainstream: creating the parallel applications and producing clusters that are easy to use.

To create parallel applications for parallel processing, the existing approach is predominately via manual methods. To be beneficial, the design and implementation time must be comparable or less than existing sequential program development. Our research into parallel application design and development has found a common trait. Despite advances in assistance to the programmer (through languages, libraries and semiautomatic tools) there is no solution that does not require the programmer to have the necessary skills to create a parallel application.

Once created, parallel applications also need to be executed. Firstly, for the cluster environment a virtual machine must be setup for parallel execution. Secondly, the parallel application must contain the appropriate directives for the execution environment to instantiate and execute the parallel application. Whilst the directives used in many environment are relatively simple, the programmer still needs to have knowledge of what to insert into the parallel application and where - not an easy process.

In this thesis we investigate the practicability of designing and evaluating an automated parallelisation creation and execution tool. Firstly, such a tool removes the requirement of the programmer to provide input into how a program should be parallelised. Secondly, the establishment of a virtual machine, insertion of execution environment specific directives and the instantiation/execution of the parallel application are not required with the proposed tool.

Automated parallelisation is synthesised via a parallelising compiler. The parallelising compiler carries out the parallelisation task automatically, requiring no extra ordinary input from the programmer. The approach taken to perform the
automatic parallelisation, is to use modular phases. Each phase concentrates on one
element central to the overall parallelisation process, generating information to pass
onto the next phase. Therefore, the compiler is built from the ground up with a
flexible and modifiable internal structure. However, our approach is different from
those used in parallelising compilers, as it addresses not only the identification of
units of parallelisation, but also information and execution directives/primitives that
are passed onto an execution environment.

Execution of parallel applications generated by the parallelising compiler is
synthesised by integrating (or linking) the parallelising compiler and the execution
environment. The link is formed when the primitives provided by the execution
environment are inserted into the parallel application. The information required by
these primitives is defined by their design and implementation. The primitives used
are made available through previous research independent to this research. The
information they require correlates to the servers of the execution environment. The
link between the two components is vital for the automated execution component of
the tool and therefore a completely automated tool.

The tool has been implemented to utilise the GENESIS cluster based
execution environment. The parallelisation component is run on the UNIX system
which then instructs GENESIS to execute the application. GENESIS is a microkernel
and client-server based operating system. The services it provides to assist parallel
processing include: dynamic process migration, distributed shared memory, reliable
communication protocols (both unicast and group based) and a parallel management
system. This execution environment is demonstrated as appropriate for the automated
parallelisation tasks requested.

To test whether the tool is able to produce parallel applications and execute
them automatically, we use qualitative and quantitative experiments. These
experiments provide both proof-of-concept and proof-of-performance. We present
our approach to testing, and how any challenges were overcome. Two particular
aspects of the parallelisation capabilities of the tool (recursion and nested loops) are
examined. For both, an examination of the parallelisation carried out is presented,
followed by the performance of each under a variety of various operational
characteristics. For both of the test cases, the expected parallelisation was carried out successfully and good speed increases for the execution environment were observed for the characteristics available.

The automated approach to parallelisation and execution that this research has succeeded in providing enables parallel processing to be adapted much quicker than traditional manual methods. The results of the tool vindicate the unique design, in particular the exploitation of resources using a efficient execution environment.
Chapter 1 Introduction

The current trend to run parallel applications is to use non-dedicated clusters (of PC’s or more powerful workstations). However there are three major obstacles that hinder making parallel computing on clusters part of the computing mainstream: finding and expressing parallelism in applications; managing efficiently the available parallel processes and computational resources (parallelism management); and making clusters easy to use by offering services transparently and automatically. To solve these three issues, sound research into this area is needed.

1.1 Motivation

There are a few methods programmers can adopt to increase the speed of and decrease the execution time of an application. Some of these are: use faster hardware, optimise the application to achieve greater efficiency, select a more efficient operating system, or parallelise the application.

The first method requires resources that a lot of programmers do not have at their disposal. The second requires expertise in optimisation which can mean extra training. The third demands extra expense that may not be available (or skill and time to write a more efficient operating system themselves). The forth method implies that extra training is needed too.

The benefits of each on their own may be substantial, but through combining two or more in an automated tool would allow the programmer to gain the best results. Combining parallelisation with an efficient operating system (or execution environment) is one step to more efficient execution. Automating the parallelisation and execution is the most ideal of them all.

Finding and expressing parallelism can be performed manually, as has been accomplished in the vast majority of applications; or automatically, using a parallelising compiler [LIAO99]. The former can lead to good parallel applications that are executed in a short time. However, it is very time consuming, error prone and expensive.
Parallel languages and tools are valuable in assisting the programmer to develop parallel applications. The one major problem with them is that the programmer needs to know several things. One, how the application being parallelised is structured. Two, the resources available within the execution environment. Three, how to program in a parallel manner. This places a large burden on the programmer that does not need to be. Automation of parallel application development is a much better solution because the programmer does not need to know anything new for the sequential program to be parallelised.

Automatic parallelisation, despite huge research efforts has not achieved fully satisfactory results [PADU96]. More promising results have been achieved using hybrid models [KWON99]. Current parallelising compilers are responsible for the identification of units of parallelism [HALL95], including quite complex and time consuming data transformations [BACO94], which allow further and quicker parallelisation to be performed.

Automated parallelism is not a particularly easy task. Many different approaches have been studied and reported on SUIF [WILS94], Parascope [COOP93], Polaris [PADU93], Parafrase-2 [POLY90] and Paradigm [BANE95]. The majority of these works have not been implemented into a fully automated system. Also most require subsequent decisions on the parallelism to be made by the programmer [MOUR99].

There are many different parallel and distributed languages and libraries in use today. Some of the most common are: Linda [CARR89], ORCA [BAL91], PVM [GEIS94], MPI [DONG96], Distributed C [PLEI93], Split-C [CULL93], and HPF [FOX90]. All of these provide approaches for parallel processes to communicate with each other efficiently. The methods used for this communication varies from shared memory to remote IPC. The programmer needs to take this into account when developing a parallel application, something that takes some time and requires additional skills.

The approach taken by PVM (Parallel Virtual Machine) [GEIS94] and MPI (Message Passing Interface) [KWON99] executing on existing operating systems, is
to provide primitives that extend and enhance the operating system that produces services for supporting parallel processing. The PVM and MPI forms of manual parallelisation require the programmer to have an extensive knowledge of the parallel software package being utilised, and also the configuration of the environment which will execute the parallelised application. Programmers not familiar with the software must spend time learning these new techniques in order to utilise the processing power of the parallel execution environment.

To streamline the development of parallel applications, a parallelism model can be followed. A chosen model effectively categorises an application into a class that defines scalability. The SPMD and MasterWorker parallelism models are inherently scalable as they allow multiple processes to compute on various data. The granularity of the execution environment (in terms of communication latency) is the primary consideration that needs to be made.

The classification of processes into a particular level of granularity is useful in determining the best execution environment for a particular application. The granularity of a process is assessed by determining the amount of time a process is performing computations as opposed to communications.

The amount of communication between parallel processes is one of the crucial issues when deciding on the type of execution environment to use. If there is high level of communication between parallel processes, then the granularity is fine. The opposite case is coarse granularity. Therefore communication latency is a critical issue when deciding on an appropriate execution environment. Applications that are used within distributed environments are much more concerned with the cost of communication than applications executing on massively parallel environments. This is because each processing unit must communicate with another over a network which has a higher cost of communication [CAP94].

This brief report shows that parallelism management, which is managing parallel processes and computation resources, is still in its infancy. Currently, programmers must deal not only with communication between, and coordination of, parallel processes, but also with managing parallel processes of individual
applications and cluster resources. Problems addressed by parallelism management include: establishment of a virtual machine, mapping processes to computers of the virtual machine, processes instantiation, load balancing and coordination of the execution of parallel applications on the virtual machine [GOSC00].

The logical step to be taken after creating the parallel applications is to execute them. To do this an appropriate parallel computer system must be used. Thus a suitable execution environment is necessary. The requirements of an appropriate environment must be drawn out first. The requirements revolve around the type of parallel application that is to be executed. In the case of this research coarse grain applications following the SPMD model define broadly what architecture should be used, but there are also requirements for the initialisation and execution of the parallel application.

Cluster based execution environments provide scalability of processing power at a relatively inexpensive cost. Scalability is important when considering an execution environment for parallel processing. The parallel applications generated can have varying demands on execution resources. Therefore, the execution environment must be able to adopt a variety of execution demands.

Examples of such cluster based execution environments include Beowulf [RIDG97], NOW [ANDE95], MOSIX [BARA98] and GENESIS [GOSC01]. These systems support the creation of single processes and static process allocation (with the exception of MOSIX and GENESIS that support load balancing), performance is only acceptable and transparency has been neglected. Further flexibility of services to support the programmer and parallel application are needed. These services include establishing and mapping processes to computers as well as static and dynamic process management. When providing these services it is important to emphasise transparency. The GENESIS execution environment also provides efficient communication methods, dynamic process allocation, process management and high level shared memory services that are all focused on providing a transparent solution - providing an exception to other cluster based execution environments.

Automating the creation of parallel applications and providing parallelism
management would save considerable time and make parallel processing more transparent. However, even if parallelisation and execution problems were solved successfully, ordinary programmers (such as an engineer, accountant, manager, researcher with basic programming familiarity) would experience many issues to be able to take advantage of the outcomes. [GOSC97] suggested that a parallelising compiler should not only parallelise an application but should also generate information for and provide some information to an operating system for controlling initialisation and execution of parallel processes.

1.2 Research Aims

The aims of this research are:

- to develop new technology that provides automated parallelisation and execution of parallel applications,
- to demonstrate feasibility of this technology by building a tool that combines a parallelising compiler with a parallel execution environment, and
- to show that the tool is usable, in particular easy to use by ordinary engineers, managers, etc., and requiring no extra ordinary input from the programmer; and is able to improve execution performance.

To achieve these aims three groups of tasks need to be addressed:

1) Synthesise a fully automated parallelisation and execution tool as the first part of the “proof-of-concept” task, in particular:
   - design (at the logical level) a compiler that provides automatic parallelisation and generation of information for a parallel execution environment.
   - incorporate parallelism models that are suitable for the source application and cluster based execution environment.
   - identify services that could provide automatic execution of parallel applications based on the outcomes of the parallelising compiler.
2) Implement the automated tool to complete the “proof-of-concept” task.
3) Carry out the study into the “proof-of-usability” and
“proof-of-performance”, including:
- research the interaction between the parallelising compiler and the execution environment.
- show the practicality of the tool synthesised in the first group of research tasks.
- demonstrate performance of the execution of parallel applications.

1.3 Research Methods

The aims of this research are addressed using the experimental computer science approach [SNYD94]. Experimental computer science is well documented as a plausible and beneficial approach to research [TICH98]. The development of a new technology that combines a parallelising compiler and a parallel execution environment involve many different aspects of computer software development, which cannot be completely analysed theoretically.

In the context of experimental computer science, this research uses proof-of-concept, proof-of-usability and proof-of-performance methods [SNYD94], [PLAI95]. The design and implementation issues of an automated parallelisation tool addresses the proof-of-concept, and the feasibility of the tool acts as proof-of-performance and proof-of-usability.

1.4 Overview of this Thesis

In order to demonstrate that the aims of the research have been achieved, the thesis is structured in the following manner. Chapter 2 presents the background issues and related work in the areas of parallel application creation and the execution of said applications. The review reveals a lack of research on the link between automatic creation and automatic execution. A programmer is still required to manually link the two elements together. From this point, research into further improving the information generated by parallelising compilers so that a completely automated approach can be developed is proposed.

Chapter 3 presents the synthesis of an automated parallelisation and execution
Chapter 1 - Introduction

Such a tool consists of a parallelising compiler and an execution environment. The parallelising compiler is responsible for performing analysis on a sequential program with the main construct used in this analysis being a *block*. The analysis allows the parallelising compiler to decide on how to parallelise the sequential program and produces the parallelised application and other information to execution environment. The decision to parallelise is based on the SPMD and MasterWorker parallelism models. The information generated by the parallelising compiler is then used by the execution environment. Requirements of what the execution environment needs in order to execute a parallel application and what is provided to the parallelising compiler are presented here. Finally, the overall design of the automated parallelisation tool is proposed and justified.

The feasibility of parallelisation and execution tool is presented in Chapter 4. The implementation of the automated tool is described along with the cluster operating system, GENESIS. The automated parallelisation and execution tool uses GENESIS as an efficient and transparent execution environment, relieving the programmer from any administrative cluster oriented tasks. The structure of the environment, the services used within the tool and the way in which the parallelisation and execution tool interacts with GENESIS is elaborated. The interaction between the execution environment and the parallelising compiler is presented also. This chapter concludes with a description of the testing environment used to evaluate the feasibility of the tool.

Chapter 5 presents the usability benefits of the automated parallelisation and execution tool - these being that the programmer is no longer required to write a parallelised application from scratch, and manual management of parallel processes is no longer needed. Furthermore, the performance study of two basic applications is shown. Details of the test environment are explained. The test programs are presented to show the variance of the parallelisation tool. The results obtained are compared against the original sequential version of the program. Finally the results and the influence both GENESIS and the type of source program has on the parallelisation process are discussed.

The final chapter summarises the thesis and concludes that the aims of this
research have been achieved. The future directions for this work are also presented.
Chapter 2  
Related Work and Concepts

2.1  Introduction

In Chapter 1 we proposed the development of a new technology that provides automated parallelisation and execution of parallel applications. This technology must be easy to use, relieve programmers from the error prone and time consuming parallelisation and execution process and be able to provide efficient execution of parallel applications. The programmer must not be required to provide extra ordinary input for both the creation and execution of a parallel application. In order to synthesise such a tool an appropriate survey and review of current technologies must be conducted.

In this chapter we present an overview of the related work in the area of parallel application development and execution of parallel applications. This chapter is structured as follows. In Section 2.2 we examine the ways and means parallel applications are developed. In particular, the characteristics of automated parallelisation and tools/techniques currently being used. This is followed by reviewing the execution environment aspects of parallel application execution in Section 2.3.

2.2  Parallel Application Creation

There are two principal techniques used in parallel program creation. The first technique is to develop a parallel program manually by employing libraries or languages. Some examples of the former are PVM [GEIS94] and MPI [DONG96] and of the latter are Linda [CARR88], Distributed C [PLEI93], ORCA [BAL92], Split-C [CULL93] and HPF [HPFF94], [HPFF97]. The second technique is to develop a parallel program automatically, using automated parallelisation tools. Automated parallelisation tools can be either semi-automatic (sometimes visual tools) such as CAPTools [IERO96], GPE [CALI99], PTOPP [EIGE93] and MERLIN [KIM00], or fully automatic that exploit parallelising compilers (Paradigm [BANE95], Parafrase-2 [POLY90], Parascope [HALL93], SUIF [WILS94]).
The manual techniques listed above all require the programmer to learn either another language, or a library to gain the benefits of parallelism. PVM for instance, requires the programmer to learn new parallel processing primitives as well as how to setup the PVM virtual machine. As for the benefits to parallel processing, these tools do not provide all aspects of parallel processing that are required and possible (e.g. process migration, load balancing, shared memory mechanisms, etc.) In fact only basic streamlined/simplified IPC mechanisms are provided [ROUG99].

This means that the programmer has to become an expert to utilise these techniques. The programmer must also structure the parallel application to take advantage of a particular execution environment, otherwise the benefit from one execution to another is not realised. When deciding on the execution of a parallel application, the programmer must make sure that resources are utilised most appropriately. Basically, the programmer must become part of the operating system!

To transfer away from this traditional manual approach, assisted or a fully automated solution must be adopted. Several projects exist that provide automated parallelisation to some extent (some are listed above). This section presents the characteristics and specific implementations of existing automated parallel application creation techniques.

2.2.1 Assisted Semi-automatic Parallelisation Techniques

The creation of parallel applications automatically is ultimately the ideal solution to the complexities that burden the manual techniques developed. These complexities include the parallel analysis the programmer is required to perform, decisions about how the parallel application is to be instantiated and the execution environment specifics that are best suited to the parallel application. The techniques that are used within automated parallelisation must be presented first before a survey of technologies can be presented.

Analysis

To discern if a sequential program or application is parallelisable some form
Chapter 2 - Related Work and Concepts

of analysis must be carried out. The analysis identifies sections of the source application that can be parallelised. Many different techniques can be used for this, such as the those presented in Part II of [WILK99] and [MOUR99].

In an ideal situation, the source program can be divided into independent parts that can be executed simultaneously without much effort at all. An application of this type could possibly have no dependencies between the parallel parts. Therefore, it would obtain a perfect speed-up as more computers or computation node are added [WILK99].

When this ideal is not met, there is most probably some dependency between the parallel parts and therefore some data distribution and collection is required. A systematic and common approach to solving parallelisation when dependencies exist is to use a master slave approach [WILK99]. This approach follows closely the Single Process Multiple Data (SPMD) parallelism model. The SPMD model is appropriate for coarse grain parallelism. The individual parallel processes developed when following this model focus on computation rather than communication.

**Synchronisation**

Data dependency between sections of code is important as synchronisation is required between parallel components so that there is no data corruption. A simple form of synchronisation is a critical section. A critical section involves mutual exclusion, which could be implemented using semaphores that allow any process to protect a region of code such that only one process can be active at a time. This is usually managed by the process requesting a semaphore (shared value accessible by all processes) which is kept until processing of the critical section is complete. At this point the semaphore is released for all processes again [DIJK68]. This form of synchronisation is important for data dependent parallel processing at a low level but does not guarantee the order of accesses [WOLF96b].

Another form of synchronisation is a barrier [WOLF96b]. Barriers are just that, barriers. When a parallel process reaches a barrier, it waits until the barrier is released. By using this entity a larger parallel processes can be created that contain
multiple sections of computation, where each section is divided by a barrier. The benefits of this are a saving of the overhead of handling separate processes being created and exiting all the time [WILK99].

**Automation**

Parallelising compilers are specialised compilers that examine sequential programs and extract any possible parallelism out of that sequential program, generating an appropriate parallel program [WOLF96a]. According to [WOLF96a] the parallelisation task can be split up into three sub-problems. These are: identifying potential parallelism; mapping the parallelism onto the target machine; and generating and optimising parallel code. The first of these sub-problems can be performed either manually, through explicit constructs placed by the programmer, or automatically through using a parallelising compiler. The second sub-problem is specific to the operating system that is controlling the execution environment. The third sub-problem is oriented towards the execution environment architecture as a whole.

Most of the research that has been carried out in association with parallelising compilers has been theoretical [LILJ94], [YANG95], [GIRK95], [FAHR97], [STOH97] to name a few. The theoretical results, whilst promising, have not been adopted widely. For parallelising compilers to be more widely utilised by the everyday computing community, more work on the practical implementation of these theoretical views must be carried out.

2.2.2 **Survey of Assisted and Semi-automatic Parallelisation Tools**

In the world of assisted and semi-automatic parallelisation there are many different tools, some of these are CAPTools [IERO96], GPE [CALI99], PTOPP [EIGE93], MERLIN [KIM00] and the outcome of work by [ARMS98]. Parallelising compilers attempt to automate the parallelisation process further, some of the most widely used are discussed here, namely ParaScope [COOP93], SUIF [WILS94], Polaris [PADU93], Parafrase-2 [POLY90] and Paradigm [BANE95]. These compilers have been chosen for the breadth of coverage of the various parallelisation models, analysis and transformations that are performed.
2.2.2.1 ParaScope

The ParaScope Editor (PED) [HALL93, COOP93] is a visualisation tool that allows the user/programmer to interact with the creation of parallel programs. It has been developed at the Centre for Research on Parallel Computation (CRPC) at Rice University (Houston, Texas). PED provides assistance in parallelising FORTRAN applications by combining programmer expertise with extensive analysis and program transformations. The analysis and transformations performed provide valuable insight into the possible parallelisation that can be extracted from an program. For PED to be beneficial the programmer is required to know how parallelisation works and an in depth knowledge of the application being parallelised.

The placement of parallel processes also poses a problem for the shared-memory model used with this tool. The FORTRAN language used with the shared-memory does not provide any mechanism to allow the programmer to specify data placement. This is overcome through using an extended FORTRAN language, FORTRAN D [FOX90].

ParaScope formed the starting point for the D System. The D System is a suite of tools that support parallel application development with the FORTRAN D language. The parallel programs that the Parascope Editor generates are primarily for fine-grain parallelism.

2.2.2.2 SUIF

The SUIF parallelising compiler is being developed at Stanford University (Stanford, California) [HALL96]. It is a collection of programs and libraries that combine to form a compiler. The compiler contains several passes that perform a variety of transformations. Some of these transformations are: parallel loop detection, parallel code generation, cache-based loop transformations, scalar optimisations, etc. [WILS94]. SUIF performs two main parallelisation operations, data dependence analysis and to some extent loop level parallelisation. The loop transformations performed are for coarse-grain parallelism. The SUIF Explorer builds on the infrastructure of the SUIF compiler and provides visual indications to the programmer.
that reduces the programmers required knowledge [LIAO99]. The whole SUIF parallelising compiler system provides a good foundation for semi-automated parallelisation.

The parallelism model adopted by SUIF is SPMD (Single Program Multiple Data) for shared memory and distributed address space machines. The passes of the compiler are built on top of a kernel that defines the intermediate coding format. Each pass performs a single analysis or transformation for the parallelisation process. The one unfortunate aspect to the implementation of the SUIF parallelising compiler is that after each phase the output is saved to a file then read in before the next phase starts. This obviously is less efficient than keeping the output from each phase in memory and passing it onto the next phase, but allows for easy rearranging of the phases to find the most optimal parallelised output.

2.2.2.3 Polaris

Polaris has been developed at the Centre for Supercomputing Research and Development (CSRD) at the University of Illinois (Urbana-Champaign, Illinois) [BLUM96]. The Polaris parallelising compiler attempts to automatically parallelise FORTRAN 77 programs, with the primary execution environment of shared-memory multiprocessors. The input language of FORTRAN 77 requires some directives that allow the user to specify explicit points of parallelism. As with the SUIF parallelising compiler, Polaris also performs several passes, each pass performing valuable tasks. Some of these tasks are: array privatisation, data dependence testing, induction variable recognition, inter procedural analysis, symbolic program analysis, etc. The implementation of the Polaris parallelising compiler is based on Delta [PETE92]. Delta was created to prototype, develop and test new source-to-source transformations for parallelising compilers that supported the FORTRAN 77 language. The only downfall with Delta itself was that it was not practical as a production compiler [PADU93].

Polaris is primarily designed for massively parallel machines. It generates explicitly parallel code that will be generic enough so that it can exploit parallelism on a variety of global address space massively parallel processors (MPP’s). Internally,
Chapter 2 - Related Work and Concepts

Polaris is composed of an abstract syntax tree on top of which exists many layers of functionality.

2.2.2.4 Parafrase-2

The Parafrase-2 parallelising compiler is also being developed at CSRD at the University of Illinois (Urbana-Champaign, Illinois) [POLY90]. Parafrase-2 is a parallelising compiler which has been implemented as a source-to-source restructurer. It can handle both the FORTRAN and C procedural languages. As with the other two parallelising compilers already mentioned, Parafrase-2 also performs passes during the compilation process. Some of these passes are: code analysis, transformations, code generation, etc. The parallelism tests that Parafrase-2 performs are quite comprehensive and is one of the most advanced parallelising compilers that performs any loop-level parallelisation, with most of the checking performed symbolically. This parallelising compiler also provides a graphical interface to allow the user to see the data dependence graphs. This is quite useful as it would be left up to the user to construct these graphs before the parallelisation points could be verified. Parafrase-2 is a very useful tool for developing new techniques in parallelising compiler design, as it provides many of the basic data structures needed to implement new ideas without the overhead [POLY90].

Parafrase-2 is designed for distributed memory multiprocessors. Although similar to the compilers built and used within shared memory massively parallel architectures, this parallelising compiler provides greater flexibility in its design so that compilers built for other architectures are able to adopt their methods. Parafrase-2 relies heavily on symbolic manipulation of the source code, commonly called symbolic dependence graphs [POLY90]. This is emphasised by [BANE95], stating that “many compilers in the future will rely on off-the-shelf symbolic packages”.

2.2.2.5 Paradigm

Parallelising compiler for distributed-memory general multicomputers (PARADIGM) exploits data parallelism and functional parallelism through combining compiling methods and run-time library support [BANE95]. Paradigm
began its development at the Centre for Reliable and High-Performance Computing (CRHPC) at University of Illinois (Urbana-Champaign, Illinois). It is now being developed at the Centre for Parallel and Distributed Computing (CPDC) at Northwestern University in Evanston, Illinois. The Paradigm parallelising compiler uses Parafrase-2 as a preprocessor that parses the sequential program into an intermediate representation or symbolic form. This entails analysis of the code and generation of flow, dependence and call graphs [BANE95].

2.2.3 Review

In sections 2.2.1 and 2.2.2 we have presented an assessment of the various automatic parallel application creation mechanisms. In particular, we have examined the assistance that a programmer is provided and existing tools (Parascope, SUIF, Polaris, Parafrase-2, Paradigm). For parallel application creation, the parallel applications created are designed for massively parallel processor (MPP) execution environments. The nature of the MPP environments is defined by a high speed bus between computational nodes. This high speed bus means that latency between processes on separate nodes is very low. Therefore the influence of communications is negligible and suited to fine grain parallelism.

All of the parallelising compilers surveyed accept FORTRAN (or a dialect of FORTRAN) as an input language. Some even accept the C language as well (SUIF and Parafrase-2). In fact the SUIF parallelising compiler converts a FORTRAN program to C first then performs the parallelisation. The reason FORTRAN is the “choice” language used by all the compilers is that they are designed for massively parallel shared memory architectures, which FORTRAN suits well and are historically from a physics background (which is where FORTRAN was developed). These languages cater for common procedural programming skills that a programmer will readily have. Therefore, as part of our research a similar approach must be taken to assist the average programmer. By using common procedural languages, such as FORTRAN, C or Pascal to be used as the source language for compilation the programmer will not have to learn any new languages.

The approach used by some of the parallelising compilers presented above
require additional directives to assist the compilation process. Whilst this may extend the amount of parallelism that could be extracted from a sequential program, the programmer would need some knowledge of parallel application development. Therefore to achieve the aim of this research, the programmer will not be required to provide any extra input to create a parallel application with our tool.

All of these parallelising compilers show that theoretical approaches to automated parallelisation can be implemented. The parallel applications created do still require the programmer to provide input into the source program and also assisting the instantiation of the parallel application. The quality of the generated parallel applications is good, but not as good as a specialist human programmer. This is because the analysis a parallelising compiler can perform is limited by the design of the compiler itself. For example, if an application uses a bubble sorting algorithm the parallelisation would be restricted to some basic loop transformation. But if a human was parallelising the application, then a quick sort algorithm could be used. The partitions of the quick sort algorithm could then be used as the parallel components, therefore producing a better parallel application.

Some of the advances generated through the development of these compilers benefit the parallelising community greatly. All these compilers adopt some form of data-level parallelism tests, particularly those designed primarily for shared memory MPP’s. Functional or task-level parallelism is a less commonly used technique, which is more commonly associated with distributed environments (or clusters) and is one of the aims of this research. The association of task-level parallelism and clusters is due to the granularity of the cluster environment and the larger computation created by a task.

The most common construct examined in the generation of parallel programs through a parallelising compiler is loop-level parallelism, since it is the most predominant source of parallelism in scientific applications [GIRK95]. A loop allows the SPMD computational model to be utilised very well and is used within all the reviewed compilers. Bacon et al. [BACO94] provides an extensive survey of loop-oriented parallelisation and vectorisation techniques currently in use. These techniques are invaluable when developing a parallelising compiler.
Of the compilers discussed above, all perform some form of loop-level parallelism analysis, mostly through data dependence testing. Except for Paradigm, none of these parallelising compilers perform task-level parallelism analysis. This is due to the prevalent and historical combination of the MPP execution environment and the FORTRAN language. Even though Paradigm is an exception, it only exploits task-level parallelism to a small degree.

As for task-level or functional parallelism, this research was unable to find any widespread use of a parallelising compiler that satisfies the needs of such parallelism. The focus of the compilation element of our research is functional parallelism and therefore ties in well with the cluster based execution environment.

The parallelising compiler tools discussed above all focus on fine grain parallelism as the parallel applications are generated for MPP shared memory architectures. This research aims to utilise cluster based execution and therefore a coarse grain parallel application is required. Whilst a loop based parallelism model as utilised by the SUIF parallelising compiler could be modified for a coarse grained environment, a functional or task-level approach would be better suited. A theoretical approach to functional or task-level parallelism by [GIRK95] states that their approach can be applied to any level of granularity. This task-level approach is adopted by the Parafrase-2 parallelising compiler.

Previous work by [EVAN97] has examined task level parallelism and developed the concept of a block to characterise a task that could possibly be parallelised. Our research aims to utilise this block based approach to extend functional parallelism and therefore using it to satisfy the coarse grain parallelism that is required by the chosen cluster based execution environment.

The concept of multiple phases is a common approach to compilers in general and in particular parallelising compilers [AHO86] and [LIAO99]. All the parallelising compilers discussed here use this phase approach. It is a good technique because each phase has only one task to perform and therefore enables easy re-ordering of parallelisation checks and optimisations, during development, to generate the best results. In the case of the SUIF parallelising compiler the latter phases can be re-
arranged to find the optimal arrangement so that the best parallel programs can be achieved.

This review has allowed us to characterise current efforts towards automated parallelisation. Many of these efforts provide some parallelisation of an application, but the onus is on the programmer to make sure an appropriate parallel application is generated in the end. This research aims to complete the process by focusing on completely automating the parallelisation without the programmers input. Thus, not every parallelisation technique can be used and is not the focus of this research. The techniques that will be used must focus on actually creating parallel applications and generating information that allows a parallelised program to be executed automatically.

To execute the generated parallel application the specifics of the execution environment are usually applied when generating the parallel application. The programmer is responsible for making sure that the appropriate mechanisms from the execution environment are available and are used. To provide a completely automated service, the parallelising compiler must use the services of the execution environment via appropriate mechanisms as part of the generation process. The execution environment must also provide services that support parallel processing in an efficient and transparent way. This is elaborated further within the following section.

### 2.3 Execution of Parallel Applications

Even if the parallel application is created completely automatically, it is of no use if the execution environment is heavily dependent on the programmer. There are many execution environments that exist providing a means for parallel processing but all require the programmer to provide instructions on how to operate.

To achieve the aims of this research, the programmer dependencies that exist in parallel process execution must be overcome. We wish to develop a completely automated parallelisation creation and execution tool. In particular, a method to provide automatic execution is to be developed. Automated execution can only be carried out if the parallel application is linked tightly to the execution environment.
To achieve this, the execution environment must provide services that are required by the parallel application.

Current technologies such as Berkeley’s NOW [VULL97], Beowulf [RIDG97], MOSIX [BARA98], Amoeba [TANE91], Sprite [OUST88], V [CHER88], Chorus [ROZI92], Globe [STEE99] and GENESIS [GOSC01] are successful in providing many parallel processing mechanisms to the programmer. The core approach taken by these technologies is to utilise commodity PC’s. In particular Beowulf and Berkeley’s NOW use a middleware type solution on top of Intel based computers/workstations or SunSPARC workstations. MOSIX is an enhancement of the Linux operating system (which is based on a monolithic kernel) use to construct a distributed operating system and hence still uses commodity PC’s.

The assistance to parallel programming and processing provided by these technologies is presented within this section as a review of the services and characteristics required, followed by a survey of current solutions.

### 2.3.1 Services and Characteristics Required for Automatic Parallel Processing

To carry out parallel processing upon a cluster automatically, the existing manual procedures and characteristics (or requirements of the programmer) must first be identified. The automated solution can therefore be synthesised from this information. Our research has found the following common requirements of clusters: suitable coarse granularity, automatic load balancing, efficient and transparent message passing communication mechanisms, shared memory mechanisms, process instantiation, automatic process mapping to computers upon instantiation and synchronisation mechanisms. These services must be presented in a manner that is efficient and transparent for the programmer to use them.

**Granularity**

The granularity of parallel applications that are executed on a cluster based execution environment are inherently coarse grained. This is due to the high communication latency between computational nodes that exists in a cluster.
environment. As newer network technologies emerge these latencies are decreasing, but are still relatively larger than those of a bus based massively parallel processing system. Therefore the method of partitioning parallel applications for a cluster is important. The granularity of the partitioned parallel application should follow an appropriate granularity for the chosen execution environment [YANG99]. An ideal situation would be for the design of a parallel application to be flexible in its granularity, so that an optimal solution could be found, but that is not always the case [WILK99].

A parallel application developed for a cluster based execution environment must be oriented towards a coarse granularity. The role of the execution environment is to balance the cluster so that the amount of remote communication between parallel processes is kept to a minimum. Load balancing techniques that allow processes to be moved between workstations must keep this communication requirement at the top of its design priorities.

**Parallelism Management**

To make sure that a parallel application is able to exploit the execution environment fully, the operating system of the execution environment must provide a comprehensive parallelism management system [HOBB00]. A parallelism management system is responsible for the initial placement of parallel processes as well as continually balancing the load of each workstation/computer. The mechanisms required to carry out parallelism management are: process instantiation, process mapping, process migration, system monitoring mechanisms.

Process instantiation is critical in establishing parallel processes. The parameters required by the execution environment must be both concise and transparent. Process mapping is used when a process is initially instantiated to automatically map the newly created process to a computer. Without such a mechanism the programmer is left to specifying the placement of processes or relying on a simple allocation algorithm that some middleware provides (e.g. PVM and MPI).

Process migration can be used to assist in balancing the load computers within
Chapter 2 - Related Work and Concepts

the cluster. The balancing of computers is important, as the resources of the cluster must be used wisely to gain the greatest benefits. The traditional representation of the load of a system is based on the average number of processes waiting for the CPU over a given time period. Within a cluster, this value along with communication load is also important. Communication load is based on the communications between two or more remote processes. The larger the number of messages (not necessarily the size of the messages) the higher the communication based load. A load balancing mechanism that encompasses both types of load for the balancing decision is important - particularly when high latency networks are involved.

These process management services should be provided without the programmer explicitly requesting them. The programmer should only have to specify the one primitive to create or instantiate each parallel process. That one primitive should not have any location requirements, in other words is should be transparent.

Efficient Message Passing Mechanisms

When parallel applications are written, communication between parallel processes is essential to distribute data necessary to carry out the appropriate computations. For effective and efficient communications, a well structured and standard mechanism is required. The message passing model (or paradigm) can be used for this.

When the programmer generates the code to send information from one parallel process to another, the information required by the execution environment must be concise and simple. A concise and simple message passing interface is critical to support the programmer. Common libraries such as PVM and MPI are intermediate solutions, but require porting to each new execution environment that is developed.

The naming of processes is critical in assisting the programmer. Commonly the individual location and local identifier (specific to each particular execution environment) are required. Therefore to simplify the tasks of the programmer, the name of end points to communicate must be transparent. This means that all specifics
Chapter 2 - Related Work and Concepts

of a communication endpoint - including the location of a process the communication protocol, etc. - are hidden from the programmer. This can be managed by either a message passing based library (external to the execution environment) or the execution environment itself.

The content of the message must also be easily stored for transmission. Complex storage methods make the programmers life harder. To simplify the data packaging of a message, a generic buffer mechanism where any type of data can be passed to the message passing service without conversion should be provided.

Finally, the underlying transport mechanisms used to transmit the messages should be as efficient (in terms of the utilisation of the available bandwidth) as possible. To achieve this efficiency a combination of unicast and multicast (or group communications) can be used selectively. When a request is made to transmit a message, the name identifying the destination entity could be a single process or group of processes. The execution environment must be responsible for choosing the most appropriate mechanism for transmission based on this naming.

Shared Memory Mechanisms

To provide a higher level parallel programming paradigm, for larger and more sophisticated parallel applications, support for the shared memory paradigm is required. Shared memory allows the programmer to remove the repetitive and specific message passing sections of a parallel application and insert references to shared memory instead. This implies a need for the execution environment to take on the shared memory management responsibility and therefore must provide the services to do so.

To provide transparent access to shared memory, the method used to instantiate processes should automatically attach each process to a common shared memory region. The management of the shared memory region including synchronisation is then the responsibility of the execution environment.
Chapter 2 - Related Work and Concepts

Synchronisation Mechanisms

One of the most important components of parallel processing is synchronisation. Without synchronisation a parallel application would not be able to execute appropriately. Synchronisation allows the programmer to ensure that correct gathering of data occurs in a non corrupted manner. This means that the results generated by a parallel section of the application must be brought together in a certain sequence so as to not corrupt the overall results. Synchronisation comes in three forms. Firstly, via a barrier. Secondly, via semaphores where mutual exclusion of a critical section is required. Thirdly, using the parallel application structure to wait for exiting processes.

For barriers and semaphores a shared resource is required. For a barrier, all the processes involved with the barrier need to register (or signal) that they have reached the barrier, before execution can continue. A barrier allows the programmer to design a single parallel program that performs several tasks and is able to synchronise between each task. When creating several separate parallel programs which constitute the one parallel application, synchronisation can be performed when each parallel processes exits. The same outcome is achieved, as with the single parallel program with barriers, but the overheads are higher as there are several processes to create from different files and cleanup after termination. Therefore barriers are a more efficient mechanism to use in that regard.

When a critical section is identified and implemented in a parallel program, all processes that are created from that program need access to the shared semaphore. The easiest way to implement these two mechanisms is via a distributed shared memory system, but message passing could be used also. Deadlock is an issue that must be dealt with when using semaphores via message passing, which is resolved within shared memory and therefore shared memory is often a simpler choice.

The waiting form of synchronisation is commonly provided by any multi-tasking operating system for a parent processes to clean up after any child processes have completed their calculations and have exited. This technique can be used in a structured way to synchronise a parallel application, particularly when the
Chapter 2 - Related Work and Concepts

MasterWorker parallelism model is being used.

2.3.2 Survey of Execution Environments

The cluster approach to parallel and distributed computing has strengthened in recent years, especially with the decreasing cost of commodity hardware. Dedicated clusters are in the main stream and are the simplest and perhaps the cleanest to implement. By cleanest we mean that the hardware and software that are used to create a cluster is uniform (i.e. bulk purchase of 100 commodity computers with a common operating system).

A selection of dedicated clusters are discussed in this section, namely Beowulf [RIDG97], Berkeley’s NOW [VULL97], MOSIX [BARA98] and GENESIS [GOSC01]. These particular cluster execution environments have been chosen as a representative sample supporting the attributes and mechanisms presented in Section 2.3.1 to varying degrees.

2.3.2.1 Beowulf

The Beowulf project [RIDG97] was established in January of 1992, sponsored by the NASA High Performance Computing Systems Group (HPCC) and The California Institute of Technology (CALTECH). The aim of the project is to use commodity PC’s together to get as many gigaflops\(^1\) as possible. The project arose from the need to manipulate large data sets by end user scientists. The challenge of high latencies for accessing data from a file server is addressed by this project.

The term Pile-of-PC’s is created by [RIDG97] to describe a loose ensemble of PC’s working together upon a single problem. The emphasis on market commodity components, dedicated processors and a dedicated local area network (or more accurately a system area network - SAN) substantiates the Pile-of-PC’s term. This varies from other approaches that focus on reclaiming cycles from idle workstations in a non-dedicated cluster.

---

\(1\) The term gigaflops represents \(10^9\) floating point operations per second.
One of the design philosophies of Beowulf is that it is freely available for the wider community to utilise. In line with this philosophy, the Linux and BSD (Unix/POSIX) operating systems are used as a base to work from. The reason that these operating systems have been chosen is the performance and availability of each system are abundant.

Within Beowulf, the operating names of processes use a global process identifier scheme that is provided for use internally and externally through a library. The internal use is independent of any external libraries. The external use is compatible with the task id of the PVM library, and is also handled transparently by traditional UNIX system calls, such as `kill()` and `getpid()`.

The granularity of the Beowulf dedicated cluster is coarse grain as the latency between nodes is relatively high, particularly as faster processors are used with existing network technologies. The Beowulf topology allows flexible growth but each new node must be manually allocated as resource allocation has not been dealt with.

The parallel programming interfaces that Beowulf supports are PVM, MPI and the Block Synchronous Parallel model (or BSP [VALI90]) an abstract parallel programming layer. Below these interfaces there is no support for dynamic process management which is desired. This means that programmers are required to develop, setup and execute parallel application on Beowulf. The mapping of processes to nodes is not supported automatically, which requires the programmer to either map processes statically or use the facilities within a library such as PVM. Regardless as to the method used to map processes on Beowulf, the processes execute on only the allocated nodes until completion.

Beowulf has no explicit support for message passing or shared memory. The only message passing provided is implicitly through the PVM and MPI libraries, which once again require the programmers input to implement and utilise. Shared memory has not been examined by Beowulf either and as such no synchronisation mechanism is offered. Synchronisation is not explicitly provided for by Beowulf either, but common techniques that use PVM and/or MPI can be used.
Chapter 2 - Related Work and Concepts

2.3.2.2 Berkeley’s NOW

The Berkeley NOW system [CULL97], [ANDE95] is being developed at the Computer Science Division of the University of California, Berkeley (hence its name). Berkeley’s solution to high performance parallel computing is to use a Network of Workstations (NOW). The concept is not too dissimilar to the Beowulf project in that it aims to provide inexpensive, low latency, high bandwidth, scalable interconnection distributed systems (or clusters) [CULL97].

The software used to coordinate the NOW cluster is layered. Each node (an UltraSPARC) executes a complete Solaris Unix installation which includes the process management, memory management, file system, thread support, scheduler and device drivers for each node. The centralised operating system on each node is extended via a middleware layer called GLUnix which provides the global operations over the whole cluster. This layer is implemented via sockets, daemons and signals. The global operations include some rudimentary process management facilities. These facilities provide a global shell with UNIX familiar kill and stop (CTRL-C and CTRL-Z) command line controls. Manual ps and kill UNIX utilities are globalised too, through the GLUnix layer [GHOR98].

A unique feature of this cluster is the global file system (called xFS) which stripes blocks of file over nodes in a RAID like manner. The dynamic approach to the low level file blocks allows workstations that are idle to act as a pseudo file server. This ensures that the best performance is achieved. The most valuable aspect of the design of Berkeley’s NOW environment is the complete operating system and file system on each node.

The messaging system within NOW is based on active messages, which are essentially remote procedure calls. The active messages are used to utilise the high speed communications available. On top of this infrastructure lies an implementation of MPI, which utilises the Active Message operations as the abstract device interface layer of MPI or the Split-C [CULL93] parallel language. The GLUnix layer provides a network process identifier (NPID) which is effectively a UNIX pid at a global (i.e. cluster) level. The NPID is used as a handle to communicate with processes via
traditional UNIX signals. The standard UNIX redirection of file descriptors is also provided at the global level through the GLUnix layer.

The NOW project at Berkeley has provided an avenue to solving many unique issues related to clusters, particularly with a high bandwidth infrastructure. However, the services provided for parallelism management are limited. To instantiate parallel processes a semi-transparent method is provided with mapping to nodes based on the load of each node. Once mapped, there are no load balancing services within NOW to balance the cluster whilst parallel applications are being executed.

The higher level shared memory paradigm is not provided either. On the other hand there is support for the common resource of files. File access is supported in a distributed manner, catering for parallel applications with a high level of file accessing and files of a large size, but if this is not the case, then the file support is of little use.

Synchronisation in the form of barriers is provided via the GLUnix layer, but there is no semaphore support for mutual exclusion of critical sections of an application. Traditional synchronisation of processes where the parent process waits for one or more child processes to exit is provided.

2.3.2.3 MOSIX

The MOSIX system is developed at the Hewbrew University of Jerusalem, Israel. MOSIX [BARA98], [BARA99] has been developed and redeveloped seven times. In its most recent incarnation, the Linux operating system is used as its basis into which the MOSIX specific services have been placed.

The architecture of MOSIX consists of multiple nodes working cooperatively as if they are part of a single system. Each node may be a single processor workstation (in [BARA98]’s words, collections of “share nothing” workstations) or multiple processor workstations (SMP’s).

On top of the nodes, MOSIX consists of a set of algorithms that support
adaptive resource sharing over the cluster and pre-emptive process migration. Within
the Linux version of MOSIX, the implementation of these two components is within
kernel level loadable modules. This allows the MOSIX layer to be completely
transparent to the parallel application level.

Whilst the parallel management decisions of a MOSIX cluster are automatic
they can be overridden by the user, who can migrate processes manually. In either
case, the migration facility allows the maximum performance of the cluster to be
obtained.

As with other clusters and their distributed nature, there is no central
controlling or master-slave relationship between nodes. Instead nodes work in pairs to
facilitate the resource sharing algorithms, load-balancing and memory ushering.
Load-balancing is based on the load (i.e. the number of processes waiting for a time
slice of the CPU over a given period of time). Memory ushering is a technique used to
place (or map) processes upon workstations (or computers) that have adequate free
physical memory (i.e. RAM). This is an attempt to prevent excessive paging to disk,
which is the case if there is a shortage of free physical memory. Because of the impact
of excessive disk accessing (i.e. paging or swapping), memory ushering takes priority
over load-balancing.

Whilst migration is provided, the underlying mechanism splits each process
into two contexts: a user context and a system context. The user context (containing
the program code, stack, data, memory maps and registers of the process) is the part
that is physically moved from one node to another. The system context is fixed at the
home node of the process (the instantiation point) [BARA99]. The system context
contains a description of the resources which the process is attached to and the
kernel-stack for the execution of system code on behalf of the process. The one
drawback in this two contexts approach is that each system call induces overhead to
instruct the system context and receive the response. This can be overcome by
strategic initial placement when programming with PVM/MPI.

In a similar fashion to the execution environments presented above in Sections
2.3.2.1 and 2.3.2.2, MOSIX relies on the programmer to create and place processes on
particular nodes (via PVM middleware). However the load balancing and memory
ushering provided by MOSIX ensure that parallel applications are able to execute as
efficiently as possible.

Shared memory is not supported by MOSIX, instead the individual nodes of a
MOSIX cluster manage memory individually. Any sharing of memory which can
facilitate support for a shared entity like a semaphore is not provided. Synchronisation
mechanisms such as an exit and wait combination are provided but the barrier type of
synchronisation is not provided.

2.3.2.4 GENESIS

The GENESIS system [GOSC01] is being developed at Deakin University,
Geelong, Australia. The approach taken for parallel processing on clusters with
GENESIS is from the ground up. The foundations are based on a distributed operating
system (DOS). The DOS consists of a microkernel and kernel servers, in a
client-server relationship.

The kernel servers facilitate the services GENESIS provides to the
programmer. The services can be classified into two groups: the first group, related to
management of basic operating system resources such as processes, memory, IPC and
files. The second group are parallelism management services [GOSC00]. Each
service is responsible for its own task and nothing else. The services cooperate to
provide the programmer with a transparent environment. This means that some
services may be implicitly accessed without the programmer being aware of accessing
them.

The current hardware base for the GENESIS cluster is legacy Sun 3/50
workstations with a slow 10Mbps network. This homogeneous environment is
currently being ported to the familiar Intel™ platform for greater scalability and cost
advantages.

Despite the hardware limitations of the GENESIS project, the software
support is far greater than that provided by other systems. The focus of the kernel
services are to *support* parallel processing. This includes parallel application development support through simple message passing library calls or PVM [ROUG97].

Active support of processes is provided by a parallelism management system [HOBB00]. The parallelism management system employs the GENESIS migration service to transfer processes around the workstations of the GENESIS cluster. Unlike the split migration service of MOSIX, GENESIS’ migration service is complete, where all elements of a process are migrated to a remote workstation therefore reducing the overhead of a split system. The parallelism management system also assists with initial process placement. This initial placement relieves the programmer from having to worry about the resources of the cluster. Once a parallel application has been established, the parallelism management system constantly monitors the cluster and balances the cluster using the migration service. Within the other execution environments above there has not been any that provides both services to the programmer.

As well as traditional message passing mechanisms for the programmer to use, distributed shared memory (DSM) [SILC98b] is also available. This service enables the programmer to adopt a higher level of parallel programming. This removes cumbersome tasks such as setting up explicit messages between parallel processes.

Traditional synchronisation of processes is achieved via an exit and wait combination, similarly to the other execution environments examined in Sections 2.3.2.1, 2.3.2.2 and 2.3.2.3. Extended synchronisation is also provided via DSM within GENESIS which includes barrier based synchronisation and mutual exclusion of a critical section with semaphores. These synchronisation services are provided to the programmer via simple primitives that only require a shared memory variable representing a barrier or a semaphore.

**2.3.3 Review**

In Section 2.3.1, we presented an overview of the various required
characteristics and mechanisms of a cluster based execution environment. In Section 2.3.2 we presented a survey of the representative cluster based execution environments that attempt to satisfy the requirements.

We found that the focus of almost all cluster execution environments is the communication cost between computational nodes. With the exception of MOSIX and GENESIS, the solution adopted is to increase the bandwidth of the network and therefore reduce the communication overhead. The network technology predominantly adopted is the Myrinet architecture and topologies, albeit a very specialised and not particularly cheap network to use. The overhead of communication must be kept to a minimum if high performance is required and communications are high within a parallel application. Even with high bandwidth communications and low communication overheads, the parallel application must still match the granularity of the execution environment for the best performance to be gained. For a cluster based environment a coarse level of granularity is therefore a requirement [GOSC97].

The support given to the parallel application developer for each execution environment varied from very little to the main focus of the environment. The support required includes: assisting the programmer with development of the parallel application, initial placement of processes to workstations and management of processes when they are executing. Most distributed environments provide a suite of parallel execution primitives with varying degrees of transparency in their invocation. For example, of the surveyed execution environments, Beowulf, Berkeley’s NOW and MOSIX provide execution on top of other operating systems, such as Solaris, Linux, FreeBSD, etc. The modifications to these environments provide familiarity and some transparency to experienced programmers. GENESIS provides its own platform for development with its own library of commands; some similarity to the UNIX programming environments exist. On top of the execution environments, all of the surveyed systems provide a middleware layer in PVM or MPI implementation, that provides the programmer with a familiar environment to use. Despite this, the programmer must either have knowledge of the PVM tool or be skilled in parallel application development to utilise these environments.
Irregardless to how the parallel application is developed, process instantiation is an important issue. Individual processes must be mapped to workstations in a simple and efficient manner. The PVM environment allocates processes via a round robin scheme, but the initial workstations that can be used must be specified. In all of the surveyed environments except GENESIS, the workstations must be specified explicitly by the programmer/administrator responsible for the setup of the cluster. GENESIS gathers this information automatically without the programmer providing any workstation information. This is also the case for parallel applications developed for GENESIS that do not use the PVM library.

The methods of instantiation found to be commonly used amongst the reviewed execution environments are: a spawn method, a twin (or duplication) method and a creation method. The spawn method of process instantiation was found to be used by the PVM middleware library. With this method an image on a disk is instantiated upon specified computers. The twin method of process instantiation required a process to be executing first. When the twin method is invoked, a duplicate process is established and execution continues from the point of duplication in the now, two processes. This has the benefit that multiple parallel sections of a parallel application can come out of the one program, but with execution environment implementations based around the unix \texttt{fork()} and \texttt{exec()} combination of primitives, the destination location of the twinned process is not flexible. The creation method of process instantiation was found to be more flexible in the implementations reviewed. The creation method is similar to the spawn method and allows a remote located to be specified. In the case of MOSIX and GENESIS, the location is not even required, as automatic allocation (or mapping) is provided.

Once the parallel processes are executing, the balance of the cluster must be maintained so that the best processing throughput can be achieved. This means that each processor of each workstation must be kept as busy as possible, but not overloaded with processes or high remote communication. These optimisations must be kept constant throughout the execution life of a parallel application or applications. MOSIX and GENESIS were the only two environments that provided load balancing services. The primary mechanisms used to move active processes in each environment was process migration.
In all, the Beowulf and Berkeley NOW systems provide only streamlined interprocess communication mechanisms, global process identifiers and limited memory management support. MOSIX and GENESIS improve on this and provide process management which relieves the programmer even further. Furthermore GENESIS provides global memory management via DSM and provides further transparency, only requiring the bare minimum specific details that are required for a parallel application to be executed.

2.4 Summary

In this chapter we have examined the current automated parallel application creation techniques and cluster based execution environments to execute parallel applications. Each have been reviewed against a set of characteristics that have been determined from background research. The characteristics focus on facilitating total automated parallel creation and execution of a parallel application and will lead to the requirements of this research.

To achieve the best performance of parallel processing, current trends lead to skilled programmers adopting tools to assist in the parallelisation process. The outcome of this is a parallel application that is a solution to the initial problem being solved. The parallel application is then executed on a cluster which the programmer must establish in order for the parallel application to execute appropriately. Simplifying this process and relieving the programmer is a must to attain more widespread adaptation of parallel processing on clusters.

This research has found that existing parallel application creation techniques using parallelising compilers still rely on the programmer having extensive knowledge of the application being parallelised and also of the parallel execution environment that the parallelised program is to be executed on. In some cases the programmer must also specify whether particular sections of the program can be parallelised or not. As well, the parallel applications generated from these existing parallelising compilers are usually only suitable for massively parallel shared memory machines. That is, they are suited to a fine grain execution environment. To remove the programmer dependency, the extra input required of the programmer must be
removed from the equation.

The existing automated parallelisation techniques found as part of this research are all well designed and developed. The common failing is that the tools that are implemented to utilise these techniques do not relieve the programmer from having to know something or a lot about those techniques. This research aims to utilise the current techniques in a parallelising compiler form. More importantly remove the requirement that the programmer needs to know anything about those techniques.

For the execution of the generated parallel application an appropriate execution environment is required. Within this chapter several cluster based execution environments were reviewed and several similar concepts were found. All the environments are designed for execution of parallel applications created by a programmer. Support is given to the programmer, either by a middleware layer such as PVM and MPI or lower level message passing and shared memory primitives native to the execution environment. During the execution of the parallel processes comprising the parallel application, support varies over the reviewed environments from no support to dynamic load balancing. The desired execution environment must provide support to achieve the most efficient execution possible; efficient is in terms of both the parallel application and the resources of the execution environment.

The one problem discovered when using such a loosely coupled environment for the execution of parallelised applications is that the communication costs are very high and bottlenecks can therefore occur [CAP94]. The granularity of the parallelised applications must match the execution environment which for the loosely coupled environment is coarse grained. This means that the parallelisation of an application must use the granularity as a constraint on the communications required.

This research has found that there is currently no transparent link between creating a parallel application and executing the parallel application. To be able to provide this service, the requirements of both sides need to be met. This means that the parallel analysis performed must be designed with the destination execution environment in mind. The destination execution environment must also provide
mechanisms (i.e. libraries and process management services) to support the generated parallel application.

Performance studies of the reviewed systems are few and far between. Other distributed and cluster based systems that provide performance data use freely available data for common algorithms of a computational nature. These algorithms would be useful to compare, but the differences of the revised systems at the operating system environment and architecture level are not readily comparable to each other. Especially when a valid quantum to measure in this area of research is the overall execution time.
Chapter 3  Synthesis of an Automated Parallel Application Creation and Execution Tool

3.1 Introduction

The aim of this chapter is to report on the synthesis of an automated tool for parallel application creation and execution on a cluster.

From the programmers point of view, specifying individual components of parallelisation manually is one approach to parallelising applications. This can lead to parallel programs that miss obvious parallelisation. It could also cause an “over” parallelised program. Apart from this, the programmer must also have an in-depth knowledge of the problem and program to be able to carry out this parallelisation. An alternative to this method is assisted automation where an analyser produces solutions, which require the programmer to select the “most suitable” version of the program. However this approach also requires the programmer to have good knowledge of the parallelisation process.

When it comes time to execute a parallel application, the programmer can also be required to provide input about the execution environment characteristics. This includes specifying where individual processes are to be executed, managing the processes when the execution environment becomes unbalanced, and handling the termination of processes. These characteristics place further burden on the programmer which does not have to be the case.

A better approach is to provide a tool that does not require anything extra from the programmer. Such a tool would be able to perform various analysis and data transformations to create a parallelised version of the application. An automated parallelisation tool forms a “black box” that takes a sequential program, parallelises it, sets up an efficient execution environment and executes the parallelised program. The results at the end should be exactly the same as if the original sequential program was
This chapter is organised as follows. In Section 3.2 the requirements of the tool are presented. The synthesis of the tool is presented in Section 3.3. Finally, a summary of the chapter is presented in Section 3.4

3.2 Requirements of the tool

The goal of this section is to identify the requirements of an automated parallel application creation and execution tool. There are three major sets of requirements. First of all, there are requirements of the programmer. Secondly, there are parallelisation process requirements, and thirdly there are execution requirements.

3.2.1 Requirements of the Programmer

To parallelise a program manually, the original program needs to be split up (or partitioned) into components that will eventually execute in “parallel”. Parallelisation alone is quite a complex process, and requires both an in depth understanding of the problem, sequential program and extraordinary programming skills. These skills include choosing a parallel model which satisfies the program/problem at hand (these models are presented in Section 3.3.2.)

Existing systems supporting manual parallelisation, such as MPI [DONG96] and PVM [GEIS94], provide language primitives to the programmer that allow manually written parallel applications to utilise parallel execution environments. The MPI and PVM methods follow a message passing paradigm that the programmer must apply when developing the parallel application. An alternative to this is a memory sharing paradigm that allows parallel processes to share memory space between each other, allowing parallel processes to “communicate”. In either case, the programmer is required to do all the work in developing the parallel application.

The parallelisation method chosen by the programmer may be based upon either a message passing or shared memory approach to create a parallel application. For a message passing approach the programmer is commonly required to specify
Chapter 3 - Synthesis of an Automated Parallel Application Creation and Execution Tool

every aspect of the communication between processes, instantiating the processes and also balancing the load of the execution environment [GOSC00]. For a shared memory approach, information about the initialisation of processes that share memory is required as well as specifying synchronisation between processes so that the shared memory space does not become corrupted [SILC98a].

A parallel application also needs to be executed in a special execution environment. The execution environment can be a modified form of an existing operating system (such as Unix/Linux or Windows). Such an execution environment must provide extra services to facilitate efficient and transparent execution of parallel applications. The extra services should include: process mapping for location transparency, parallelism management and process termination for execution transparency [HOBB98], [GOSC00]. These services are required so that the parallel application can be executed and consist of initialisation of the execution environment for and management of parallel processes. The programmer must also setup the execution environment, no matter what sort it is.

The common approach to utilising an execution environment for parallel processing involves the programmer in one way or another. The initialisation of the execution environment entails setting up the communication and parallelism management servers (process instantiation, load balancing, termination, etc.) on each computational node within a parallel execution environment. The management of parallel processes is handled by both the parallel application and servers within the computational node. To establish such an environment, knowledge of the components and setup of the execution environment is required, which places an extra load on the programmer. An example scenario of using a manual approach is shown in Figure 3.1. The programmer is obviously involved in many stages of the creation and execution of a parallel application using manual tools.

The programmer should not be involved in these execution environment oriented tasks. The execution environment should be able to coordinate parallel processes on behalf of the programmer and needs to continue for the life cycle of each parallel process. Existing manual parallelisation methods, such as MPI [DONG96] and PVM [GEIS94], require the programmer to initialise the execution environment.
Further information about the configuration of the execution environment is required as processes are created and messages sent.

When the programmer executes a program the execution environment must provide several characteristics [GOSC01]. These characteristics are: performance, ease of use and transparency. The performance must be the best available, this involves cost and speed. A cluster based execution environment should be comparable to the performance of super computers. The execution environment must be easy to use, requiring little or no input regarding which individual computers to use within a cluster. The programmer should be unaware of the location of which a parallel application is executed (i.e. the specification of the location is transparently carried out) and the communication between processes.

In summary, the programmer must perform several tedious, cumbersome and sometimes daunting tasks to produce a parallel application manually. To begin with the parallel application needs to be developed and written. This includes inserting the appropriate directives to make the execution environment perform in the appropriate manner and the parallel application actually execute. To make sure that the parallel application is executed as efficiently as possible, the coordination of processes that constitute a parallel application is critical. Often this is left to the decisions made by the programmer. An example is process to computer mapping, the execution environment should perform this mapping automatically. Also, once the processes are executing the balancing of the execution environment must be maintained at all times.
so that the desired efficient execution occurs.

### 3.2.2 Automated Parallelisation Requirements

To satisfy the programmer’s requirement “to create a parallel application automatically”, a facility that provides this service - an enhanced parallelising compiler - is needed.

A parallelising compiler extends the concepts of a traditional sequential compiler to provide a parallel application as output rather than a sequential application. As with all traditional compilers, the output produced is low level code for a particular target architecture.

The benefit of a compiler, either a traditional or parallelising compiler, is that the work of translating the source to the output is performed by the computer without any input from the programmer. With a parallelising compiler the translation must include some analysis that is able to identify, partition and generate parallel components of the original sequential program.

The structure of a traditional compiler is to follow a set of phases [AHO86]. Each phase performs some analysis which generates information which is passed on to the next phase (Figure 3.2) where it is used as a building block. Each analysis performed follows well documented techniques [ASAK97], [BACO94], [REPS95], [WILS95].

![Figure 3.2 Phases concept](image)

All phases of analysis are *invisible* to the programmer. The information that is passed between phases is internal to the parallelising compiler and is therefore not seen by the programmer. The information consists of symbol tables and syntax trees.
In particular, the structure of the syntax tree contains flags that are used to signal when parallel sections begin and end, synchronisation points, communication information, etc.

The objective of separating the analysis into phases is to provide a modular system. This allows well defined phases to be implemented without obfuscating the whole parallelisation process. As a whole, the phases combine to provide the most efficient parallelised program possible. This means that both the design of the parallelising compiler and the overall execution time of the generated parallel program are efficient.

The requirement of the phases as a whole is to break down the source program into parallel components that can be eventually executed in parallel. To assist this process a parallel model can be mapped to the source program. This allows standard parallel processing mechanisms within the execution environment to be utilised once the parallel program has been created.

To provide effective parallelisation, the parallelising compiler must be able to distribute data appropriately. This is what enhances the parallelising compiler to provide a totally parallelised application. The way in which data is distributed depends on the parallelism model chosen. Two types of parallelism, data and functional, may be utilised within the parallelisation process, both specify the partitioning used in the parallelisation process. These two types allow the most efficient parallelisation to be obtained.

Common parallelism models that are used are: Single Program Single Data (SPSD), Single Program Multiple Data (SPMD), Multiple Program Single Data (MPSD) and Multiple Program Multiple Data (MPMD) [WILK99]. Each focus on different aspects of the distribution of data and functionality of a parallel application. However within our research the SPMD model of parallelism is the focus as the execution environment is cluster based and is therefore well-suited.

Within the SPMD parallelism model, both the data and functionality within an application need to be partitioned. Data parallelism partitions by data, which means
that all processes execute the same program, operating on different data. The SPMD model is well-suited to problems with regular, predictable communication patterns [MOUR99].

Functional parallelism, by definition, partitions by tasks. Each process performs a different function (routine) or executes a different section of the program. “MasterWorker” [FOST95], [CORN00], [WOLF96b] is a modification of the SPMD model. With this model, a single program (Master) coordinates the work performed by all the Worker processes. The Master may or may not contribute to the overall computation of the program. The MasterWorker model is an example of functional parallelism because each Worker process may perform different tasks. The data and functional models of parallelism are not mutually exclusive and can therefore be used together. Within this research both the SPMD and MasterWorker models are to be used.

For both data and functional parallelism models, the data can be partitioned automatically by a parallelising compiler during the analysis phases. Within these phases, the data used to perform calculations is known, and in some cases is the reason for the decision to parallelise. The best parallelism outcome can be achieved through a combination of data and functional parallelism.

Data is spread amongst the parallel processes via the analysis performed on the sequential program. Part of the analysis is to decide which data is sent to which process. It is this decision-making process that constitutes the heart of the parallelising compiler component. Once the program is parallelised, the output contains the data to be distributed to each process.

An example of the method of data distribution is matrix multiplication (see Figure 3.3). In this case nested loops are commonly used to solve the problem. These nested loops are detected and partitioned so that each row could in fact form a basis of a parallel process. Each parallel process is then responsible for the calculations for a particular row over all columns. As the results are calculated, and the parallel process exits, the results are returned to the parent process.
Chapter 3 - Synthesis of an Automated Parallel Application Creation and Execution Tool

The scalability of the SPMD model is only restricted by the conditions put forward by the execution environment. Scalability is an important factor to take into account when deciding how to parallelise a particular program or part of a program. The conditions of the execution environment consist of the communication speed between computational nodes, memory limitations, processor speed and management of the parallel processes. The number of processes that can be created is only limited by these conditions. If too many parallel processes need to be created and each has a small task to execute, then the overhead of initialising, managing and terminating the processes will affect the overall execution time.

The concepts described above are shown in Figure 3.4. The parallel section contains either the results of the SPMD or MasterWorker model being followed. Data is presented at the start of the parallel section. This is the data derived from the analysis carried out by the phases of the parallelising compiler.
Chapter 3 - Synthesis of an Automated Parallel Application Creation and Execution Tool

The parallel models (SPMD and MasterWorker) are used as a design principle for the parallel analysis performed within the parallelising compiler. When an application is detected as fitting one of the two models, the appropriate distribution of data can be signalled within the structures of the parallelising compiler.

The parallel application created by the parallelising compiler requires the execution environment to create the specified number of processes and assist the distribution of data to and from each process. At the design stage of the parallelising compiler, the number of processors within the execution environment is not known (or required) as the execution environment is expected to handle all requests to create processes, regardless of the number. Communication between processes follows a similar requirement, the parallelising compiler requires each parallel process to be able to communicate with each other. To carry this out each process needs a communication end point. The responsible entity for handling this is the execution environment.

Therefore the requirements of the parallelising compiler are specified by a program to parallelise, a parallel model to apply to sequential source programs, information generated by each phase to pass on to the next, and support from the execution environment to carry out the instructions given in the parallel application.
The information generated by each phase contributes to the overall information passed to the execution environment once the parallel application has been created. The placement of each process to a computation node within the execution environment is crucial to the efficient execution of the parallel application. This is a requirement that can only be satisfied by the execution environment.

### 3.2.3 Execution Environment

When a programmer is creating and executing parallel applications manually, there are many required tasks to be carried out. When using a cluster based execution environment which is physically distributed with no physical shared memory, there are specific tasks that need to be completed for parallel processing to work effectively. For automatic execution of a parallel application, therefore relieving the programmer from the execution oriented tasks, the requirements for parallel processing in this environment need to be identified.

To begin with the execution environment must be initialised. This includes the initialisation of each individual computer that comprises the parallel virtual machine (VM) of the cluster. The initialisation entails booting each computer and making sure that each computer is registered within the VM. The registration is handled by the underlying distributed operating system (DOS).

Once each computer of the cluster is initialised, the parallel application must then be instantiated which is carried out by two entities, the programmer and the execution environment. A parallel application comprises of many parallel sections which means that each parallel section will need to be instantiated at some point. The common method of instantiation is creation, where an image located as a file or a file system is loaded into memory and executed. The programmer must specify the name and location of the program image, the execution environment is then responsible for the coordination of the required tasks to start executing the parallel application.

The location of the executing processes must also be specified. This is called mapping and involves each computer of the cluster at some point. The actual mapping process is important because individual computers within the VM should become
overloaded. Therefore an appropriate mapping method must be used so that overloading of individual computers within the VM does not occur.

When specifying communication points within a parallel application, the identifier or end point for a process to send information to is required. When there are numerous parallel processes within a parallel application, keeping track of these identifiers can be quite daunting.

Finally the parallel processes of a parallel application will invariably need to be synchronised as part of the overall parallel processing. The programmer must rely on the execution environment to carry out the appropriate process suspension (e.g. the parent waiting for the children) on each computer.

Therefore, when creating and executing parallel applications as a solution to a problem in a cluster based execution environment, the programmer faces several issues and difficulties. These difficulties are as follows: initialisation of the VM, instantiation of the parallel processes, mapping parallel processes to cluster computers, appropriate communication instructions for interprocess communication, synchronisation of parallel processes are carried out correctly, and load balancing for more efficient processing.

To relieve the programmer from focusing on these execution environment activities, the following mechanisms must be provided automatically and adaptively:

- execution environment initialisation
- process instantiation based on process creation
- process mapping to cluster computers
- communication among parallel processes
- synchronisation of processes
- load balancing

The execution environment that is most suitable for parallel execution is one that doesn’t require the programmer to initialise and coordinate the execution of parallel processes. Therefore an execution environment that assists (as opposed to depending on) the programmer is required. For the execution environment to be able
to assist the programmer it needs to perform resource discovery of its own to determine the resources (i.e. computational nodes) available, their state (e.g. highly loaded, heavy loaded, idle) and their parameters. With this information, the execution environment will itself know what is available to use. The information is needed to establish the VM required to execute the parallel processes.

To instantiate processes, the name (or identifier) of the program to instantiate and location are required. In some cases multiple copies of the same process are also required, and to exploit parallelism several locations need to be specified. These pieces of information should be handled automatically.

The style of process instantiation should be able to facilitate creation from an image on the file system. When instantiating the processes, the most efficient method should be used. Efficient instantiation includes factors such as how many copies of the one program image will be needed to create $n$ processes. This is relatively common place when a parallelism model such as SPMD is being followed. The location of processes instantiated must also be transparent.

Each parallel process must be mapped to a computer when it is instantiated. To overcome this the execution environment must provide facilities that perform process mapping. The resource discovery requirement (described above) carried out by the execution environment provides information that allows the servers of the execution environment to carry out the process mapping.

To communicate between parallel processes, communication end points are required. The end point is used by the execution environment as a logical entity to communicate with (i.e. a port or similar). The execution environment should be responsible for the coordination of the naming, delivery and reception of messages to these end points. Thus processes can transmit and receive via them simply without having to establish the associated transport and other lower level components that communications require.

Two sorts of communication can be used within a parallel application, they are: explicit or implicit. Explicit communication is the most common and is used
throughout parallel applications that use message passing as their communication paradigm. For example, at the end of a parallel section of code the results of computation within a child process (or processes) is sent to the parent process. Implicit communication is less commonly used and is associated with the memory sharing paradigm. This communication style is different because the individual messages used to transmit the required data is not required.

An explicit case may occur at the end of a parallel section of code, where the results are returned to the parent process. An implicit case may occur during a parallel section of a parallel application. Depending on the execution environment, one of two paradigms (message passing or memory sharing) can be chosen by the programmer to implement these communication paradigms.

The message passing paradigm requires the programmer to explicitly specify “what is being sent to where and when”. When developing a parallel application with the aim of using this paradigm, the overhead of setting up messages to send between parallel processes is an important factor to consider. Another factor to consider is the errors that are quite easily introduced, due to the amount of information that must be specified, the message passing paradigm can therefore be cumbersome to use.

With the memory sharing paradigm, the programmer does not have to explicitly specify messages. Parallel processes created under this paradigm share a memory space, and can therefore communicate via this memory. From the programmers point of view, the implementation overhead of this paradigm is less than message passing. The specifics are abstracted away from the programmer therefore making the programmers life easier. To carry out the required synchronisation when using memory sharing, appropriate synchronisation primitives must be inserted by the programmer, these primitives should be easier to use than all that is required for message passing.

When synchronising processes within a parallel application there are two states that can take place:

1) The parent process of a parallel section of code informs the execution environment that synchronisation must occur. This is usually at the end of a
parallel section of the application.

2) Each process that constitutes a parallel section of a parallel application invokes a common primitive before and after a critical section of code which provides a point at which all processes synchronise.

In the first case the values of the variables and data structures of the processes must be sent to parent process as part of the synchronisation process. In the second case, the programmer is required to place primitives around critical sections of the program. These primitives are then used by the execution environment to pause processes when the critical section is reached.

Further to the requirements of the programmer, the execution environment can also provide services or mechanisms that assist parallel processing efficiency. One in particular is parallel process management. Within a programming environment such as PVM, static and slightly assisted allocation [YU97] are the only mechanisms available for initial placement. Balancing the load of an active VM is not generally available. A balanced VM facilitates more efficient execution. Services within the execution environment should be able to move processes around so that each computational node is balanced. For this to occur, a migration service in combination with a global scheduler is required.

Therefore the requirements of the execution environment to assist automated parallelisation are:

- initialisation of the VM without input from the programmer
- process instantiation based on process creation
- automatic mapping of processes to computers of the VM
- transparent communications
- synchronisation mechanisms and dynamic process load balancing

The main emphasis placed upon these requirements consists of transparency and automation. Providing an execution environment that satisfies these requirements allows the aims of this research to be achieved.
3.2.4 Overall requirements

As separate entities, the parallelising compiler and execution environment require the programmer to coordinate them both. The parallel application from the parallelising compiler must be passed to the execution environment. The execution environment must first be initialised before it is able to execute the parallel application. The programmer is invariably responsible for joining these two entities together. This operation is performed off-line. The overall requirement of this research is to eliminate the manual components through linking the two entities together automatically.

The entire automatic parallelisation and execution process begins with the programmer specifying a sequential program, the parallelising compiler takes this program and parallelises it based on some analysis. To carry out the analysis, the parallelising compiler interprets the source application via well defined structures so that any parallelism can be detected. The parallelising compiler also inserts primitives that are native to the execution environment. The execution environment then executes the parallel application following the inserted primitives. The primitives must be self contained to simplify the work of the parallelising compiler.

The execution environment responds to the requests of the primitives through invoked services. These services support the execution of the parallel application and include process creation, process mapping, message passing, shared memory, synchronisation and process management.

3.3 Design of an Automated Parallel Creation and Execution Tool

In this section we present the logical design of the automated parallel creation and execution tool that satisfies the requirements presented in Section 3.2. To do this the architecture of the proposed automated parallelisation tool is presented. Each of the components is then discussed in detail.
3.3.1 Architecture

The proposed schema of an automated parallel application creation and execution tool is presented in Figure 3.5. This schema is used to show logically how a program is to be parallelised and executed via such a tool.

The proposed tool contains three components: a parallelising compiler, a linking section and an execution environment. These three components combine to provide an efficient and easy to use tool. Each component has well defined tasks to perform, the output from each component is passed to the next automatically. This

```
void main(int argc, char *argv[])
{
    int a;
    for(a = 0; a < 10; a++)
    /* statements */
}
```

Figure 3.5 Automated parallel creation and execution tool schema
automation removes the need for the programmer to provide any assistance throughout the whole process. The following description of the design of the automated parallel creation and execution tool presents each of the components explaining their functionality, inputs and outputs.

3.3.2 Parallelising Compiler

The parallelising compiler is responsible for the parallelisation of the sequential application as well as producing all the relevant information required so that the execution environment is able to execute the parallelised application. The structure of the parallelising compiler consists of two parts. The first part, is a traditional compiler which uses phases to carry out the various tasks of compilation. The phases include lexical, syntax and semantic analysis, optimisation and low level code generation. The second part, consists of extra phases added to the traditional compiler that perform the parallel analysis and generation of execution-oriented information required, based on information produced and collected in the phases of the first part.

The traditional compiler consists of three phases, presented in Figure 3.6. The three phases of this compiler are:

1) lexical, syntax and semantic analysis - responsible for the administrative tasks of compilation
2) code optimisation - responsible for the optimisation of statements
3) low level code generation - responsible for the generation of the low level code required for the execution environment

![Figure 3.6 Traditional compiler architecture](image-url)
To parallelise a sequential program there is a need for enhancing the traditional compiler architecture. Each phase produces information that is used in the next phase. This is achieved by incorporating extra phases being inserted between the first and second phases of the traditional compiler, as shown in Figure 3.7. These extra phases carry out the parallel analysis required and are the phases that assist in satisfying the requirements of the automated parallelisation process.

The information passed between phases consists of data structures that represent the symbols (symbol table) and flow (syntax tree) of the original sequential program. These data structures are required to allow each phase to carry out their analysis, which is specified in the forthcoming sections. After Phase 1 these two data structures are prepared and ready to be used within the next phase.

The extra phases are positioned between Phases 1 and 2, because after Phase 1 the symbol table and syntax tree data structures have been established. These data structures are required to carry out the parallel analysis. The inserted parallel phases are able to build on the two data structures. The parallel phases consist of:

1a) detection of units of parallelism - the first of three parallelisation analysis phases examines each statement of the source program for identifiers and how they relate to each other
1b) detection of sequences - the second parallelisation analysis phase examines each of the statements identified in the previous phase (1a) and groups them together into sequences
1c) insertion of synchronisation points - the third parallelisation analysis phase signals where to insert synchronisation points, depending on the information provided from Phase 1b

These three new phases are used to satisfy the parallelisation requirement of the parallelising compiler. The proposed parallelising compiler consists of six phases in total and is shown in Figure 3.7. The logical design of all phases is described in detail below.
3.3.2.1 Phase 1 - Lexical, Syntax and Semantic Analysis

The desired output from the lexical, syntax and semantic analysis phase (Phase 1) are internal structures, namely the syntax tree and symbol table. These internal structures are used to contain the flow and symbol information of the original sequential program that all subsequent phases can use to:

- traverse the syntax of the source program quickly and easily
- examine details of identifiers, including: scope, type, array index limits, value (for constants)

To produce the desired output of Phase 1 an LR(1)\(^1\) parser is proposed, as this type of parser can generate the appropriate data structures. Internally, the LALR(1)\(^2\) parsing tables are created from a language grammar that is specified [AHO86]. The parser provides the syntax tree and symbol table data structures that can be easily modified. These two data structures are then passed onto the next phase, and subsequence phases as the information required to perform the appropriate analysis is needed. An example of the syntax tree and symbol table passed from Phase 1 to Phase 2 is presented in Figure 3.8

---

1. LR(1) - a parser that employs the left-to-right scanning of input, right-most derivation in reverse with a symbol look ahead of one.
2. LALR(1) - look ahead LR parser.
Chapter 3 - Synthesis of an Automated Parallel Application Creation and Execution Tool

3.3.2.2 Phase 2 - Detection of Units of Parallelism

The syntax tree and symbol table, constructed in Phase 1, are the backbone to the analysis performed in Phase 2. This phase is the first of three phases responsible for the parallelisation of the sequential application.

To be able to detect parallelism in programs, the identification of unique segments of the source program that perform autonomous operations must be carried out. These segments are called blocks [EVAN97]. The block construct allows the compiler to group contiguous instructions together in a well defined logical fashion. Each block can then be further categorised in subsequent analysis, rather than revisiting each statement of code.

There are two types of blocks, explicit and implicit. Explicit blocks can be identified through the syntax of the source language. Implicit blocks require the parallelising compiler to identify the blocks through analysis of each statement of the source program.

At a high level of abstraction, explicit blocks in a language such as C or Pascal, can be identified quite easily. Symbols such as “{}” and begin/end, identify the start and end of blocks. These symbols represent the programmers instructions for grouping statements together. Detecting these explicit blocks is
relatively easy, as the syntax of the language lends itself to this, but the level at which
the blocks group statements doesn’t allow a wide range of parallelism to be detected.
Other units of parallelism must be identified as candidates for parallelism.

To identify implicit blocks, each statement of the source program needs to be
examined to locate identifiers and the method by which they are used. This means to
check if an identifier is being used for the value it contains (i.e. read from) or setting
the identifier to a value (i.e. written to). There are two levels of relationship between
identifiers that need to be recognized. Firstly, within each statement, the identifiers
used need to be ascertained and this information is kept in the syntax tree. Secondly,
between statements, how the identifiers are accessed between one statement and the
next must also be identified. In both cases, flags in the syntax tree are set to a
particular state depending on the access of each identifier. Other programming
constructs can also be classified as blocks, such as:

- procedures or functions
- nested procedures or functions
- recursion
- libraries

As part of the analysis of this Phase 2, these units are detected also. For this
purpose a parallel symbol table is employed. The parallel symbol table is similar to

```c
while (statements within syntax tree)
{
    /* check syntax of statement */
    find identifiers within statement
    check for
        multiple use
        write access
        read access
    note access within symbol table
    if (statement contains procedure call)
        add entry to parallel symbol table
    if (statement branches the program)
        add entry to parallel symbol table
}
```

Figure 3.9 Pseudo code to identify implicit blocks
the symbol table created in Phase 1, except that the flow of the program in terms of possible units of parallelism can be seen once the parallel symbol table has been created. The entries within the parallel symbol table allow subsequent phases to quickly confirm which units of parallelism can be utilised within a program.

The pseudo code for the identifier analysis and parallel symbol table generation is presented in Figure 3.9. The output generated from this phase and passed on to Phase 3 is presented in Figure 3.10 contains the following structures:

- modified syntax tree containing information about the identifiers
- parallel symbol table

3.3.2.3 Phase 3 - Detection of Sequences

The purpose of this phase is to decide which identified units of parallelism can actually be executed in parallel. Identifying units of parallelism is not enough to be able to execute in parallel automatically. The issue of dependency between units of parallelism and external resources must also be taken into account. If the program was left as segments identified in the previous phase and executed, then the results would be unpredictable.
A sequence is a grouping of program code that must be executed in a pre-defined order so that the values of variables and other data structures do not become invalid. Invalidity occurs when the sequences are not coordinated properly and the results produced are incorrect and do not match what would be achieved from the original sequential application. Therefore, an appropriate (or correct) ordering of the identified sequences must be used. To ensure appropriate ordering, the sequences identified must contain all the statements necessary to maintain correct computation.

To detect a sequence, dependencies between the units of parallelism or blocks from Phase 2 need to be referred to. This referral consists of the accesses to variables (including data structures) to ascertain if they are being set or read. There are three cases that can be identified:

- Firstly, two consecutive statements can set the same identifier.
- Secondly, two consecutive statements can read from the same identifier.
- Thirdly, either can read or set.

The only case that can be parallelised here is the second, as the other two have dependencies that would be broken if parallelised.

As well as detecting sequences this phase also endeavours to map the application to a parallelism model. The models used in our parallelising compiler are SPMD and MasterWorker (see Section 3.2.2). If the sequences are found to have looping (explicitly or implicitly) or repetitive statements then a SPMD model can be used. If independent parallel sections that need a coordinating entity are found, then the MasterWorker model can be used.

It may be the case with some applications that a combination of models are used, producing a hybrid parallel application. The mapping to a parallelism model allows easier transformation into a parallel application. This is because the model provides guidelines on how:

- the parallel processes should be identified
- the information should be passed to the parallel processes
- the processes are to be synchronised, and
- the information from the parallel processes is to be gathered
In terms of detecting sequences, the parallelism model assists by defining how units of parallelism can be grouped into sequences. The grouping of units of parallelism is noted within the syntax tree.

The number of processes that can be created depends on the characteristics of the execution environment and the characteristics of the relationship between parallel processes within a parallel application. The frequency of communication between parallel processes is the most important characteristic. When combined with the amount of processing time per process this is termed *granularity*.

If each parallel process frequently communicates with one or more other processes and the latency between computational nodes within the cluster is high, then the parallel application will incur a loss of performance. To solve this, either the communication between processes is reduced through a reduction in the number of parallel processes, or processes that communicate frequently are moved to the same computation node.

As was stated earlier, the communication characteristics of a cluster based execution environment are inherently slow in relation to the processing speed. In particular, the communication latency between computational nodes is quite high. This is important as the granularity of a parallel application destined for such an environment must be relatively coarse.

The granularity of the applications generated by the compiler is important when execution of the parallel application occurs. If the execution environment is not well suited to the granularity level, then the execution time will not be beneficial to the whole application (i.e. it will be slow). With this noted, the communication between processes should be kept to a minimum.

The output generated by the detection of sequences phase and passed on to Phase 4 is a modified syntax tree. As with all phases, this is passed along with the symbol table to the next phase (see Figure 3.11).
3.3.2.4 Phase 4 - Insertion of Synchronisation Points

This phase is the last of the three phases responsible for the parallelisation of the sequential application. The responsibilities of this phase are with classifying the synchronisation of the parallel sequences of the program that have been identified in Phase 3.

The sequences identified in Phase 3 need to be grouped together so that the correct results are obtained. The grouping is such that each sequence does not effect the direct results of another. If consecutive sequences do effect each other, then a synchronisation point needs to be inserted so that this does not occur.

The identification of synchronisation points is similar to the detection of sequences. In this case the checking is performed between sequences rather than within (what constitutes) a sequence. Each statement is consulted via the syntax tree, if the internal structure of the syntax tree signals a sequence, a pointer to that statement is stored. The process continues until a statement has no sequence set. At this point the previous consecutive sequences will be executed in parallel and a synchronisation point is noted within the syntax tree. Therefore, the next statement will not be executed in parallel. The pseudo code for this process is presented in Figure 3.12.

An implication of using synchronisation points is the requirement to collect data for the next sequence (or set of parallel sequences). All the values of the
variables and data structures that are modified within the parallel sections of code need to be gathered, so that the next parallel section of code can start with the correct data. This is handled by Phase 6 of the parallelising compiler. When the synchronisation point is found, the appropriate message primitives are placed into the generated parallel application.

The grouping of sequences in this phase add further to the parallelism model that is identified within Phase 3. The synchronisation points placed within the syntax tree produce parallelism that follows the SPMD and MasterWorker parallelism models. The SPMD model is followed if the sequence being parallelised is a loop. The MasterWorker model is followed when several unrelated sequences are parallelised together. The synchronisation points complete the sections of parallelism that these models define.

The output from this phase is once again a modified syntax tree which is passed along with the symbol table onto Phase 5 (see Figure 3.13). The symbol table is also passed to Phase 5 - but it is unmodified.

At the end of Phase 4 - the third of the parallel analysis phases - the information gathered and produced by the parallelising compiler is enough to specify
the primitives required to create and manipulate parallel processes. These primitives are inserted during Phase 6. The information also ensures that the coordination of the parallel processes allows the parallel application to execute correctly.

### 3.3.2.5 Phase 5 - Code Optimisation

The desired output from the code optimisation phase is an optimised program,

```plaintext
if (2 < 4) then /* statements */ ...
```

```
  if < statements
  2 4
```

```plaintext
v := 1 * a;
```

```
  :=
  v  *
  1  a
```

```plaintext
v := 7 * 64;
```

```
  :=
  v  *
  7  64
```

```plaintext
v := 7 * 64;
```

```
  :=
  v  448
```

**Figure 3.13** Phase 4 information

**Figure 3.14** Optimisation examples
where possible. The focus of this research is not optimisation of programs and therefore only minor optimisations are carried out. The optimisations are such that the parallelisation detected in the previous phases is not affected.

The types of optimisation that are examined are algebraic identities \[\text{[AHO86]}\]. Examples of these are presented in Figure 3.14. The optimisations are performed on the internal structure (syntax tree) of the parallelising compiler. A recursive routine examines the syntax tree nodes for any algebraic identities. If an identity is found, the affected nodes within the syntax tree are flagged for removal. The syntax tree is then consulted again to see if any nodes have been identified as a parallel unit prior to being identified for removal. If a node is identified for removal and is the start of a parallel units as identified by Phases 2 to 4, then the optimisation is ignored. The information generated and passed onto Phase 6 is presented in Figure 3.15.

![Figure 3.15 Phase 5 information](image)

3.3.2.6 Phase 6 - Low Level Code Generation

At the start of this phase all the parallelisation analysis has been performed. The sequences of the application that can now be executed in parallel have been identified and all that remains is the low level code generation.

Phase 6 is responsible for the generation of the low level code required to execute the program on the execution environment. It is this code that, in part, links the parallel application to the execution environment. The completed link is made
once the parallel application is executed.

To create this link, the primitives provided by the execution environment are inserted into the parallel application as the low level code is being created. For this to occur, the primitives used to carry out the required operations must be identified. The execution environment must provide facilities to support the provided primitives. The following two sections, 3.3.3 and 3.3.4 provide details of the proposed primitives and execution environment support.

### 3.3.3 Primitives

From the three parallel analysis phases of the parallelising compiler, there are several operations that are required to successfully carry out execution of the parallel application. These vary from creating a process to sending messages, each of which are accessed via primitives that are supplied by the supporting libraries of the execution environment. Each primitives must be responsible for a specific task and must also provide transparent access to the intended services of the execution environment. The transparency of primitives means that the information required is specific to the task being requested only, there is no superfluous information that is required.

### 3.3.3.1 Primitives of the Execution Environment

To provide support for the SPMD or MasterWorker models that the parallelising compiler uses, a mechanism to create several identical processes is required. Most execution environments provide mechanisms to create a single process from a program image stored on disk. Creating several identical processes can be optimised within the execution environment, if the number of processes to be created is known. The optimisations occur within the relevant facilities of the execution environment. Hence, we propose that the cluster based execution environment should provide a primitive `create_n()` to provide such optimised process creation. This primitive will require two parameters, the program image and the number of processes to be created. The location of each of the processes is allocated by the execution environment.
When a shared memory region is required, one or more processes are instantiated first and then a memory region is mapped to each process. For this to occur we propose that the execution environment provides a primitive `create_shmem()` to offer the services required to carry out this type of operation. This primitive will require the same parameters as for `create_n()`; the internal mechanisms of the `create_shmem()` primitive provide the different functionality. The location of each process is carried out by the execution environment because the functionality of the `create_n()` primitive is to be used. The `create_shmem()` primitive must also return the address of the shared memory to the created processes, so that the they are able to access the shared memory region.

When a parallel process has finished its processing, an `exit()` primitive is proposed. This primitive will ensure that the execution environment can carry out appropriate garbage collection once a process exits.

A parent process in a parallel application often needs to know when a child parallel process has completed its execution and exited. For this, a proposed primitive called `p_wait()` is proposed. This primitive will pause the parent process until a child process has exited, at which point the execution environment will resume the parent process.

When a parallel application needs to synchronise processes, the parallelising compiler chooses either, synchronisation via a type of semaphore or by pausing processes via a boundary. We propose that traditional `signal()` and `wait()` primitives [BURN90], [DIJK68] can be used for the semaphore method. Each primitive will require one parameter, a flag that is used for the semaphore operations. The facilities within the execution environment provide the synchronisation. For the boundary method, we propose that a `barrier()` primitive be provided. The parameter for this primitive will require one parameter also. This parameter is a flag, similar to the `signal()` and `wait()` primitives.

To send and receive information between processes, the message passing primitives `send()` and `recv()` are proposed. Both primitives will only require minimal information about the source and destination processes to send to and a
buffer to send from or receive to. The buffer mechanism must be generic, so that many different information types can be sent and received. Table 3.1 contains a summary of each proposed primitive and their semantics.

<table>
<thead>
<tr>
<th>Primitives</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>void create_n(program path, number of processes)</td>
<td>Instantiates number of processes copies of the program specified by program path.</td>
</tr>
<tr>
<td>addr create_shmem(program path, number of processes)</td>
<td>Instantiates number of processes copies of the program specified by program path and returns a pointer to the shared memory region created.</td>
</tr>
<tr>
<td>void exit(void)</td>
<td>Simply terminates a process.</td>
</tr>
<tr>
<td>void p_wait(process id)</td>
<td>Pauses a parent process whilst a child process is still executing. The process id parameter is populated with the process id of the child process.</td>
</tr>
<tr>
<td>void signal(semaphore)</td>
<td>Release a shared semaphore.</td>
</tr>
<tr>
<td>void wait(semaphore)</td>
<td>Wait for a shared semaphore to be released.</td>
</tr>
<tr>
<td>void barrier(barrier)</td>
<td>Wait for all processes sharing a barrier to reach the barrier.</td>
</tr>
<tr>
<td>void send(end point, message, size)</td>
<td>Send a message of a certain size to end point.</td>
</tr>
<tr>
<td>void recv(where from, message buffer, size)</td>
<td>Receive a message from where from into a message buffer of size size.</td>
</tr>
</tbody>
</table>

Table 3.1 Table of proposed primitives and their semantics

3.3.3.2 Linking Primitives into the Parallel Application

All of the primitives described in Section 3.3.3.1 instruct the underlying facilities within the execution environment to carry out the specified operations. The parallelising compiler inserts these primitives during the last phase of the compilation process. To accomplish this the syntax tree is traversed as for all the phases of the parallelising compiler, the flags of each node are checked.

The flags of the syntax tree are set by the three parallelisation phases to signify
the start/end of sequences and synchronisation points throughout the application. Each of these flags indicate different operations that need to occur during Phase 6, which in effect defines the parallel operations of the whole parallel application.

To begin with the main parallel application is placed as low level program code in a file. When the start sequence flag is found, several actions must be undertaken. Firstly, the code following that flag within the syntax tree is to be executed in parallel. Therefore that code is placed as low level program code in a separate parallel child (or slave) file. This is to be performed by traversing the syntax tree until the end sequence flag is found.

As the whole tool is designed to be executed from a command line, the input from the programmer is to be the source program and one command line option (described below) only. The essential aspect is that no extra input is required from the programmer.

When the end sequence flag is found the appropriate process creation primitive is placed into the main application. The style of creation primitives that are inserted into the parallel application can be either `create_n()` or `create_shmem()`. For the purposes of this research the selection of creation primitive is to be made from an option specified on the command line by the programmer. This selection by the programmer does not influence the parallelisation decisions made.

To create many parallel process images that are eventually instantiated, their names must be unique. If the process image names are the same, then they will cause a conflict within the execution environment. Therefore a system to generate unique names must be used. The names within a parallel application can be made unique via sequential numbering, but if more than one version of a parallel application of the same name is to be executed, then the names must also be unique between parallel applications. Therefore, intra and inter process image names must be unique.

The solution proposed for this problem is to use a name that includes a reference to the current time, represented as seconds since 1/1/1970 (commonly used
in UNIX systems as the “epoch” of time.) When each process image name is created, the current time is appended to the image name. Therefore, as time is always moving forward, the same process image name will never exist more than once. When the processes are instantiated, internal process management allocates unique id’s which remain unique throughout the execution life of the process. These id’s are independent to the unique process image id’s. The image names are used as the first argument to the create_n() and create_shmem() primitives.

Once the parallel processes have been created, they will need information for their computation. That information must either be sent explicitly via message passing techniques or implicitly via shared memory. The method used depends once again on the command line option specified by the programmer.

For either method, the information that the parallel process or processes required must first be identified. This is determined from the syntax tree once again. The statements between the start and end sequence flags are examined to find which identifiers are used. The types of these identifiers are examined by looking up the symbol table. A reference to each identifier is stored in a list, effectively a list of pointers to the syntax tree nodes, which implicitly point to the symbol table.

For message passing the send() and recv() primitives are to be used. The information is to be sent to the parallel process(es) as a structured block of data. The list of pointers is traversed to generate the appropriate low level data structures required. The order of the elements of the data structure in the parent process (sending side) must be mirrored in the child process (the receiving side) so that the individual identifier values are not mixed up. The reverse needs to be used when the child needs to send information back to the parent.

For shared memory, the same internal list of pointers can be used to generate the low level code necessary to (in effect) pass values of identifiers through the shared memory region. The shared memory region can be represented as a large data structure which is specifically generated along with the other low level code. The data structure will be the same for both the parent and child processes.
There are no specific primitives to be inserted to access the shared memory region. This is because once the shared memory region has been established any access to that shared memory region is managed by the execution environment services. The only explicit primitive that needs to be inserted is the semaphore primitives.

A semaphore is required when two or more of the parallel processes are modifying or accessing an identifier. This is determined when creating the list of identifiers as described above. If the same process is to be created more than once, the identifiers in the list are checked to see how they are accessed. If they are being modified, then the semaphore primitives signal() and wait() are inserted to enclose the critical section.

During the execution life cycle of a parallel application there will be several points at which the whole application needs to synchronise. In particular when parallel processes of a particular parallel section have completed their computation. To do this within the parallel applications generated by our tool, the parent process must synchronise with the child processes. The method for performing this synchronisation depends again on the programmers command line option selection. When message passing is specified the p_wait() primitive is to be used, with shared memory the barrier() primitive is to be used.

The p_wait() primitive needs to be placed in the parent process so that all child processes are caught when they exit. The results of the child computations need to be sent to the parent just before they exit. Once all child processes have been accounted for, sequential processing can then occur.

The barrier() primitive is placed in both the parent and child parallel processes. Similarly to the p_wait() scenario, the processes all synchronise, but do so when they reach the “barrier”. At that point the results of the child computation is passed to the parent. From this point, the child processes can exit and the parent continues as with the p_wait() scenario, or the child processes can continue execution with new information sent from the parent process after the sequential execution has completed.
Chapter 3 - Synthesis of an Automated Parallel Application Creation and Execution Tool

The algorithm used during Phase 6 described in the text above is shown in Figure 3.16. This algorithm is similar to the other phases of the compiler in the way it traverses the syntax tree.

```plaintext
set output_file to parent
while (statements)
{
    if ( start sequence flag is set ) {
        get_identifiers_used( tree );
        if ( message passing mode ) {
            /* Create the appropriate low level code to
               ** send information to the parallel process */
        }
        /* Open a file for a new parallel process */
        /* Change the output_file to this new file */
        if ( message passing mode ) {
            /* Create the appropriate low level code to
               ** receive information from the parent
               ** process */
        }
    } else if ( end sequence flag is set ) {
        if ( message passing mode ) {
            /* Create the appropriate low level code for
               ** sending information back to the parent
               ** process */
        }
        /* Close off the "new file" */
        /* Place the appropriate process creation
           ** primitive within the parent file */
        if ( message passing mode ) {
            /* Create the appropriate low level code for
               ** receiving information back from the parallel
               ** process */
        }
    } else {
        if ( within parallel process ) {
            if ( identifier is modified ) {
                /* Place signal() command */
            }
        }
        if ( not within a sequence ) {
            synchronise parallel processes
        }
    }
} /* Create low level code for statement using
   ** current output_file */
if ( within parallel process ) {
    if ( identifier is modified ) {
        /* Place wait() command */
    }
}
```

Figure 3.16 Pseudo code for Phase 6
Once Phase 6 has generated the parallel application it is passed to the execution environment. This process is shown in Figure 3.17. The graph in the box provides a visual representation of the flow of the parallel application that is passed to the execution environment.

![Figure 3.17 Phase 6 information](image)

### 3.3.4 Execution Environment

The following facilities must be provided by the execution environment to satisfy the execution environment requirements specified in Section 3.2.3 and the primitives specified in Section 3.3.3 for:

- process instantiation
- exchanging messages
- supporting shared memory
- global scheduling
- process migration

The proposed execution environment, which is able to provide these services, is a cluster based execution environment, built on top of a microkernel architecture. The microkernel architecture was selected for rapid development. In such an environment, the microkernel supports kernel servers which are responsible for
specific tasks within each computer of the cluster. The microkernel and kernel server relationship allow the operating system to utilise the available resources as much as possible. The architecture is presented in Figure 3.18.

![Figure 3.18 Execution environment architecture](image)

The microkernel and directly related servers are executed on each computer within the cluster. In the case of the Global Scheduler, the responsible server executes on one computer and acts on behalf of the whole cluster. Therefore the execution environment must consist of an operating system that controls the physical hardware of each computer and also provides services for the execution of parallel application processes.

To create the parallel processes required for the parallel application to be executed correctly, the first of the facilities listed on page 72, process instantiation, must exist. Efficient process instantiation is a top priority: when more than one copy of the same process is required, a *bulk* instantiation method is required. Therefore, there are two types of process instantiation methods required by our automation tool, single and multiple/group creation. The first method is to be used for the MasterWorker parallelism model. This is because many different processes need to be created for the one parallel section of code. When the execution environment receives a request to create a single copy of a process, the execution manager consults the global scheduler (described later) to provide a location for that process. The global scheduler replies and then the process manager at that location takes over from that point.
Chapter 3 - Synthesis of an Automated Parallel Application Creation and Execution Tool

The second method of creation is applied when an SPMD model of parallelism is being used. A request for multiple copies of the same process causes the execution environment to use a process creation mechanism. Similarly to the single process creation method (described above), the global scheduler is consulted for a location. Only this time several locations are required, the global scheduler identifies appropriate computers of a virtual parallel machine. The execution managers at each of these locations are contacted to create the relevant processes and the appropriate number of copies within. Multiple creation is used when one location is used for multiple processes. To create many processes remotely group creation is used as multiple creation does not scale well [HOBB00].

The second facility, a message passing facility, is provided via both local and remote interprocess communication (IPC) components within the cluster execution environment. The programmer requests this communication paradigm explicitly when parallel processes must pass information between each other. The basic operations for this paradigm are sending and receiving data (i.e. a process sends information using the send() primitive, another process receives the information using the recv() primitive). The data itself can be of many different formats and the location of the processes sending and receiving the information can be either local or remote. Therefore, the execution environment must provide generic mechanisms to handle this.

When two or more processes attempt to communicate with one another, the microkernel on each computer receives the message initially. It then determines if the message is destined for a local or remote process. Messages with a local destination are forwarded directly to the process via the microkernel. Messages with a remote destination are forwarded onto a local IPC manager to handle the delivery of the message. The IPC manager executes on each computer (i.e. locally) of the cluster.

The third facility of the execution environment is responsible for the provision of shared memory. This is an alternative paradigm that the programmer can use to exchange information between processes. It can also be used for process synchronisation, using for instance semaphore operations. When a parallel application requests that two or more processes are to have a share a memory region, the
Chapter 3 - Synthesis of an Automated Parallel Application Creation and Execution Tool

The execution environment must: create those processes, map the shared memory over those processes, allow the processes access to that memory and handle any updates or synchronisation required with that memory. The elements of the shared memory facility are handled by the memory manager located on each computer, in conjunction with the process and execution facilities. The request to create processes is handled by the execution manager, similarly to the creation facility above, except this time the memory manager of each computer must also coordinate the establishment of the shared memory. When an update is made to the shared memory region, the consistency of the memory amongst computers must be maintained. To achieve this the memory managers transmit the differences amongst one another.

The fourth facility, a global scheduling facility, provides services for the cluster as a whole (a distributed global scheduling facility could also be used, however its scalability is low). When the creation of a process is requested by a parallel application, the global scheduling facility provides information to the kernel servers for process placement. This placement is based on the communication patterns and load of the computers within the cluster. The global scheduler aims to balance the communications and load of all the computers within the cluster, so that an equable use of resources occurs at all times. An equable use of resources means that a parallel application will be executed more efficiently.

The resource discovery server (see Figure 3.18) must provide information about the state of each computer of the cluster. This information is fathered at regular intervals, so that a good clear state of the cluster can be identified. The information can then be used by services such as a global scheduler to make decisions about placement of processed based on the load of each computer.

When processes of a parallel application are executing, the load of one computer may become temporarily unbalanced. The global scheduler at this point employs the services of a migration management facility to shuffle processes around the computers of the cluster to re-balance the load. Migration management is provided by the fifth facility required of the execution environment. Unlike the global scheduler facility, a migration manager lies on each computer. When directed by the global scheduler, the migration manager on the originating machine performs the
migration in collaboration with the migration manager on the destination machine [DePA98].

3.4 Summary

In this chapter we have synthesised the design for an automated parallel application creation and execution tool. The requirements for the three proposed components were examined and a design was presented that satisfied the requirements. The three components consist of: a parallelising compiler, linking mechanism, and an execution environment.

The parallelising compiler design is based on a traditional compiler with information passed from one phase to the next. The information consisting of a symbol table and syntax tree provides the basis for the parallel analysis carried out by three new phases specifically designed. The three phases (Detection of Units of Parallelism, Detection of Sequences, and Insertion of Synchronisation Points) contribute cumulative decisions that are used to generate the final parallel application.

The parallel models that are used in the generated parallel applications are the SPMD and MasterWorker models. These models are used as part of the design of the parallel analysis phases to provide structure to the parallelisation/sections identified in the original sequential application. This means that the generated parallelised applications follow either of these two models of parallelism.

To complete the automation requirements of this research, the parallel application generated by the parallelising compiler is linked automatically to the execution environment. This means that the information produced by the phases of the parallelising compiler are used to insert appropriate primitives provided by the execution environment. The requirements of the programmer and the execution environment have been used in the design of both the information and primitives.

To carry out the tasks directed by the parallel application, the execution environment uses the information specified within the primitives to direct the execution environment services. The services are designed to perform the requested
tasks in an efficient and transparent manner. The execution environment has been
designed to provide services that work independently of any direct request by a
parallel application. For example, process mapping and load balancing. By providing
these services the execution of the parallel application will utilise system resources
more efficiently.

The synthesis presented in this chapter will enable a cohesive automated
parallel application creation and execution tool to be implemented. The architecture
of the whole tool (shown in Figure 3.19) is a new approach to automated,
parallelisation and execution. The combination of the three components, a
parallelising compiler, information via primitives and the execution environment
provide a concise and structured solution to a complex task.

Figure 3.19 Architecture of the whole tool
Chapter 4 Implementation of the automated parallelisation creation and execution tool

4.1 Introduction

The automated creation and execution of parallel applications is shown in Chapter 2 to be quite a complex problem. A design of a tool that automatically parallelises and executes applications that solves this problem is proposed in Chapter 3. This design includes three components: a parallelising compiler, execution environment and a link between the two. Once implemented, these three components combine to provide a transparent solution to the problem of parallelisation and execution of parallel applications that exploit the SPMD and MasterWorker models of parallelism.

The design satisfies the research aim to develop new technology that provides automated parallelisation and execution of parallel applications. The aim of this chapter is to address the feasibility of the design by building a tool that combines a parallelising compiler with a parallel execution environment. This constitutes the “proof-of-concept” [Snyder94] method used during this research.

In this chapter, we describe the implementation of the whole tool in relation to the identified components from Chapter 3. As stated in Chapter 3 (Section 3.3) the parallelising compiler consists of a set of six phases that each perform separate tasks that combine together to perform automatic parallelisation by creating a parallel application. The implementation of each phase is presented as a description of each construct that is produced and used, and the algorithms and processes followed. Included in the phase descriptions are the parallel analysis phases that identify the sections of the original sequential program that can be parallelised. These parallelisation phases form the core of the parallelising compiler.

After the internal structures of the parallelising compiler have been setup, the
low level code generation to link the parallel application to the execution environment is required. Before this can be done the selected execution environment and its supporting services must be identified. Section 4.7 of this chapter presents the architecture and services that the chosen execution environment provides. To link the parallel application to the execution environment the primitives provided by the execution environment (satisfying the design) are inserted into the parallel application during the low level code generation. The primitives and the information that is transmitted to and from the execution environment are presented in the next section of this chapter. This is followed by the process used to generate the low level code, including primitive insertion. Finally, a summary of the achievements of the feasibility study is presented at the end of this chapter.

4.2 Implementation of Phase 1 - Lexical, Syntax and Semantic Analysis

To begin with an appropriate source language needs to be chosen. The design of the compiler is such that any procedural language could be used. For our research, a procedural language called Pascal has been selected, because it provides both very strict syntax and typing mechanisms. This means that the ambiguities introduced by other procedural languages are not an issue and therefore the compilation process is simplified. The chosen language is not modified in any way from its specification. This allows a programmer familiar with Pascal to write a sequential program without having to provide any special directives.

The supplied sequential program needs to be interpreted by the compiler so that the appropriate analysis can be performed. The interpretation of the program is performed by an LALR parser. This parser has been implemented using a “grammar” for the Pascal language. Common tools for creating a parser quickly and efficiently are “lex” and “yacc” [LEVI95]. LEX (lexical analyser) examines each character in the source program and constructs tokens from these characters. YACC

2. LookAhead Left Recursive [AHO86]
3. A well defined set of rules that constitute a language.
4. The particular versions of YACC and LEX used were “bison” and “flex”, respectively.
Chapter 4 - Implementation of the automated parallelisation creation and execution tool

(yet another compiler compiler) uses the structured grammar and the tokens (generated by LEX) to reconstruct the tokens into statements, which are defined by the rules. These rules also contain semantic actions that allow the syntax of the source program to be verified. The statements created are internally represented as two data structures, a syntax tree and symbol table. It is these two structures that allow the source program to be interpreted in later phases.

The syntax tree structure is adaptable so that it can be modified during the parallel analysis phases with the relevant information required. The modifications entail flags that allow later phases to carry out their operations. The structure of the syntax tree is similar to any binary-tree. This structure is used to create the nodes that comprise the tree. The information part of the node can then be modified freely.

The syntax tree structure is presented in Figure 4.1. In this figure the bold fields are the particular entries that are modified during the analysis carried out in later phases. The ref, start_sequence, end_sequence, num_iterations, sync_point and

```c
struct node {
    int node_num;
    int operator;
    /* constants value */
    int value;
    int var_cnt;
    int param_cnt;
    char *name;
    /* the following are used within
    ** the parallel analysis */
    int ref;
    int start_sequence;
    int end_sequence;
    int num_iterations;
    int sync_point;
    int semaphore;
    OBJECT_TYPE objtype;
    TABENTRY *sym_tab_ptr;

    struct node *left;
    struct node *right;
} NODE;
```

Figure 4.1 Syntax tree structure
semaphore fields are all flags that are switched on when a condition is met. The conditions are specified in the following phases of the parallel analysis. The other fields of the structure are used in the syntax validation of the source program and during the final code generation phase to identify names and types of each node.

A separate syntax tree is created for each procedure/function within the source program. An example visualisation of the syntax tree in “tree” form is shown in Figure 4.2. When viewed in this way, each element of each statement can be easily identified. This representation is used for the development of the tool as well as allowing the reader to visualise the syntax tree.

![Figure 4.2 Syntax tree visualisation](image)

The symbol table, which is constructed within this phase, is built using a linked list. Once constructed it is not changed and is used only as a referencing mechanism. The symbol table consists of various pieces of information for all identifiers (i.e. variables and functions/procedures). The information comprises scope, object and symbol types, pass by reference, value for constants, etc. The symbol table can be visualised as a table of data, similar to a database table. An example of a symbol table is shown in Figure 4.3. Each identifier from the source program, be it variable, procedure, or function, is listed in this table. The scope and type information of the identifier allows the syntax analysis and later code generation phases to carry out their tasks.

The syntax tree references the symbol table by way of a pointer. This pointer is called `sym_tab_ptr` and can be seen in Figure 4.1. The reference is made when
Chapter 4 - Implementation of the automated parallelisation creation and execution tool

4.3 Implementation of Phase 2 - Detection of Units of Parallelism

The syntax tree and symbol table elements are the backbone of the parallelising compiler. Once these two elements have been created in Phase 1, the parallel analysis phases can next occur. Each of the parallel analysis phases examine the statements of the source program via the syntax tree using an in-order traversal. When the details of an identifier is required, the symbol table is referred to indirectly via the \texttt{sym\_tab\_ptr} (Figure 4.1) element of the syntax tree structure.

The algorithm used in Phase 2 consists primarily of a while loop that contains a large case statement. The operators of each statement are checked to see if they perform any of the following: “branch” the program, a routine call, a looping construct or an assignment. Once an operator of this type is found, the relevant elements (highlighted in Figure 4.1) of the syntax tree’s structure are modified. The elements that are modified act as flags, and therefore are turned on if the particular condition is found within the analysis carried out in this phase.

\begin{figure}[h]
\centering
\begin{tabular}{lcccccc}
\hline
\textbf{NAME} & \textbf{TMP-NAME} & \textbf{OBJ-TYPE} & \textbf{SYM-TYPE} & \textbf{SCP-TYPE} & \textbf{REF} & \textbf{VALUE} \\
\hline
MATRIX\_SIZE & & INTEGER & CONSTANT & GLOBAL & & 256 \\
GlobalMemory & & RESERVED & STRUCTURE & GLOBAL & & \\
input1 & & INTEGER & ARRAY & GLOBAL & & \\
input2 & & INTEGER & ARRAY & GLOBAL & & \\
result & & INTEGER & ARRAY & GLOBAL & & \\
glob & & RESERVED & VARIABLE & GLOBAL & & \\
row & & INTEGER & VARIABLE & GLOBAL & & \\
col & & INTEGER & VARIABLE & GLOBAL & & \\
i & & INTEGER & VARIABLE & GLOBAL & & \\
j & & INTEGER & VARIABLE & GLOBAL & & \\
sum & & INTEGER & VARIABLE & GLOBAL & & \\
first\_row & & INTEGER & VARIABLE & GLOBAL & & \\
last\_row & & INTEGER & VARIABLE & GLOBAL & & \\
ref & & RESERVED & PROCEDURE & GLOBAL & & \\
ref\_a & & INTEGER & VARIABLE & PARAMETER & 1 & \\
bob & & INTEGER & VARIABLE & GLOBAL & & \\
\hline
\end{tabular}
\caption{Symbol table visualisation}
\end{figure}
while (statements within syntax tree)
{
   /* possible units of parallelism */
   find operators within statement
   if operator is a "begin" /* sequence */
      • modify start_sequence in syntax tree for current node
   if operator is an "end" /* sequence */
      • modify end_sequence in syntax tree for current node
   if operator is an "assignment"
      • check identifier(s) on both sides of the assignment
      • identifiers are added to parallel symbol table with
         the access method stored
   if operator is a "for"
      • modify start_sequence in syntax tree for current node
      • add operator to parallel symbol table
      • modify end_sequence in syntax tree once end of the
         for loop is reached.
   if operator is a "while"
      • modify start_sequence in syntax tree for current node
      • add operator to parallel symbol table
      • modify end_sequence in syntax tree once end of the
         for loop is reached.
   if operator is a function or procedure call
      • add to parallel symbol table (as a new "flow" of
         execution)
      • check the parameters passed
         • add the parameter reference type to parallel
            symbol table
      • now traverse the syntax tree for the routine code
         and perform this whole method on this routines
}

Figure 4.4 Phase 2 pseudocode algorithm

The modifications are dependent on the type of operator that is found. In this
phase the assignment (:=) and loop (for ... begin ... end or while ... begin ...
end) operators are the focus. Other minor operators (parameters of routines,
variables, arrays, records/structures) are consulted throughout the parallelisation
process to assist in the parallelisation decisions made.

The pseudocode algorithm for this process is presented in Figure 4.4. The
algorithm shown adds entries to a parallel symbol table that is created in Phase 2 also.
This symbol table is similar to the one created in Phase 1 and is used in much the
same way. The parallel symbol table is used in conjunction with the modified syntax
tree in Phase 3 of the parallelising compiler and contains basic information about the
constructs that are added. More detailed information is obtained from the original
symbol table which is pointed to by the sym_tab_ptr element, shown as bold in
Figure 4.5 Parallel symbol table structure

An example of a parallel symbol table is shown in Figure 4.6. There are two sections in this example, separated by a blank line. The top section represents the main routine, and the bottom section represents the ref routine. The routine name is identified by the first PROC_FUNC_NAME entry of each section.

The main routine also contains reference to the ref routine that is called. This represents a call to the ref routine, which is then followed by the specific entities identified for the ref routine.

At the end of this phase the information that is generated pertains to the possible units of parallelism. The units need to be grouped into sequences so that they can be executed in parallel without corrupting information that is used within each sequence. This means that the dependencies between the units must be identified and this is the responsibility of the next phase.

4.4 Implementation of Phase 3 - Detection of Sequences

Phase 3 is responsible for identifying the parallel sequences within the source program. To do this, the syntax tree is traversed again, this time in conjunction with the parallel symbol table constructed in Phase 2.
Chapter 4 - Implementation of the automated parallelisation creation and execution tool

placed in line with the caller routine (*MAIN* of the procedure and how they are referenced. This is why the routine that is called is parallelised. If the left hand side identifiers are different then, you can always parallelise the statements. The algorithm this is shown in Figure 4.7.

The checking performed looks for two consecutive assignment operations, the identifiers being set (located on the left hand side) for each are examined. If they are the same, this is noted internally. If the identifiers on the right hand side are the same as the left then the two consecutive assignment operations cannot be executed in parallel. If the left hand identifiers are not used on the right, then they can be executed in parallel. If the left hand side identifiers are different then, you can always parallelise the statements. The algorithm this is shown in Figure 4.7.

A similar comparison is made for procedure calls, in particular the parameters of the procedure and how they are referenced. This is why the routine that is called is placed in line with the caller routine ("*MAIN*" in Figure 4.6). The routine called,
Chapter 4 - Implementation of the automated parallelisation creation and execution tool

ref, is checked to see if the parameter(s) are pass by reference. If so, then the code within the routine is most likely to modify the parameter(s). This is confirmed by checking further down the parallel symbol table. If this is the case, then that procedure is left to be executed sequentially. If the parameter(s) are not pass by reference, then the procedure may be able to execute in parallel. If some of the identifiers within the procedure are not local (i.e. global scope) then the procedure itself may not be parallelisable. The identifiers within the procedure that are global are checked against the identifiers of the next statement. If the next statement, which could also be another procedure call, requires the identifiers that are global within the procedure, then the procedure cannot be parallelised along with the next statement. If the identifiers are different, then the procedure can be parallelised. This is shown in Figure 4.8.

When a function is checked, a similar process as for the procedure is followed. The return value from the function is the only difference. In this case, the identifier being set to the return value of the function needs to be checked with the next statement to see if the function/assignment statement can be parallelised too.

For the looping constructs (for and while) the indices of the loops are checked against the body of the loop. The statements that constitute the loop body are checked for relationships between each other (similar to all the statement checked

```c
traverse syntax tree
if ( operator is assignment )
{
    if ( ptr_left != NULL )
    {
        check current left hand side identifier against
        ptr_left->identifier
        if ( the same )
            if ( right hand side list does not contains
                left hand side identifier )
                sequence flag set
            else
                sequence flag not set
        else
            sequence flag set
    }
    set ptr_left to left hand side identifier
    set ptr_right to right hand side identifier list
}
```

Figure 4.7 Identifier identification algorithm
traverse syntax tree
if ( operator is procedure/function )
{
    check routine parameters in symbol table
    if ( parameters are pass-by-reference )
        flag this within the parallel symbol table
    /* Check assignments within procedure and their
    ** modification of the pass-by-reference parameters */
    if ( any of the pass-by-reference parameters are set )
        sequence flag for procedure node not set
    else
        /* both pass-by-reference and non pass-by-reference
        ** parameters */
        sequence flag for procedure set
}

**Figure 4.8** Procedure/function parameter detection algorithm

described above). If dependencies are found to be the case, then the body of the loop
must be kept together as one block. In this case the whole loop block is used as a
parallel unit and the index of the loop is passed to that unit. When this is detected, the
num_iterations element of the syntax tree node structure is set to the appropriate
value. When this cannot be determined as a constant within the compiler, the element
is set to “-1” which is caught in Phase 6 to be handled symbolically in the low level
code generation.

During the checks described above, the syntax tree is modified when the units
of parallelism are found to be parallelisable. The start_sequence field of the
syntax tree structure (see Figure 4.1) is where the change is made. Phase 2 sets this
flag initially as a possible parallel sequence. This phase therefore only has to turn off
the flag if the above described conditions are not met.

As stated in Chapter 3, when the tool is executed a command line option is
used to specify whether to use message passing or shared memory in the form of
distributed shared memory (DSM). When DSM is specified, shared memory and its
associated primitives to create processes are used when the compiler generates the
low level code (Phase 6). Synchronisation around critical regions are able to be
implemented more easily when DSM is selected. Semaphores which are shared
through the shared memory region provided by DSM can be used. This allows the
compiler to parallelise consecutive statements that are not parallelisable when using
message passing. This further parallelisation is noted in the syntax tree by modifying the semaphore element (see Figures 4.1 and 4.9).

The output from this phase is therefore a modified syntax tree and the symbol table (unmodified) that is passed onto the next phase. The modifications arise from the algorithm used in this phase. An example of the modifications is shown in Figure 4.10.

```plaintext
traverse syntax tree
   if ( DSM mode )
   {
      /* Similar to identifier identification Figure 4.7 */
      if ( consecutive statements prove to be non-parallelisable )
         set the semaphore flag for this statement to lock
         set the other statement’s semaphore flag to lock
         set the semaphore flag of the last identifier
         before the next statement to unlock for both
         statements
   }

Figure 4.9 Semaphore identification algorithm
```

Figure 4.10 Syntax tree extract with modifications
Each box contains the modified syntax tree elements
The parallel sequences detected in this phase represent a granularity that is coarse. This follows the logical design and the execution environment characteristics.

The communications between parallel processes become prevalent in later phases, particularly Phases 4 and 6. In these phases the synchronisation and low level execution environment primitives are inserted.

The parallelism model that is observed in the selection of sequences in this phase follows the methods specified in the logical design for Phase 3. When a loop is found to be parallelisable, the SPMD model of parallelism is adopted. This means that the processes derived from the loop being parallelised all perform the same operation with different sets of data send to each instance of the processes.

The MasterWorker model is observed when two or more sequences are executed in parallel that do not perform the same operations. This almost always occurs in the parallelised program in code areas outside loops.

The syntax tree contains the relevant information for the creation of these processes. During Phase 6, the actual code to create the processes and send the data to the processes is generated.

4.5 Implementation of Phase 4 - Insertion of Synchronisation Points

Once the sequences have been detected, synchronisation between parallel sequences and sequential sequences is required. The syntax tree is traversed once again to find the sequences identified within Phase 3. This is the final phase of parallel analysis.

As the syntax tree is traversed, consecutive sequences are grouped. Each consecutive sequence can be executed in parallel, as there are no dependencies between each sequence as determined in Phase 3. Once a sequence is found that is not directly followed by another sequence, a synchronisation point is inserted. This is carried out by modifying the `sync_point` flag from the syntax tree structure (see...
Chapter 4 - Implementation of the automated parallelisation creation and execution tool

Communication between parallel processes can now be seen more clearly, as parallel units and the synchronisation of these points is known. For the two parallelism models used in our tool, the explicit communication between parallel processes and the parent process is defined in Phase 6 (low level code generation).

This phase generates enough information to know how many processes are to be simultaneously created, and what information needs to be sent to each parallel process. For example, the index of a loop will be sent to each parallel process created for a loop. Once again, Phase 6 is responsible for generating code to send this information.

4.6 Implementation of Phase 5 - Code Optimisation

The optimisation techniques used within the parallelising compiler consist of algebraic identities. As stated in the presentation of the logical design (Chapter 3), optimisations are only used when identified parallel units are not affected. This means that only statements within a parallel sequence or sequential statements are examined for modification.
Modifications take place on the syntax tree itself. As redundant nodes are found, they are removed and the tree is adjusted appropriately, shown in Figure 4.12. The syntax tree at the end of this phase is slightly smaller if optimisations are found. Therefore no explicit execution environment information is generated in this phase.

![Code optimisation example](image)

4.7 Execution Environment Services Used in the Automated Parallelisation Tool

The execution environment chosen to provide the required services of the parallelising compiler is cluster based. The services are provided upon the request of the primitives within the parallel application generated by the tool. The services and architecture of the execution environment are presented in this section.

From Section 3.3.4, the specific facilities required by our tool are: resource discovery, global scheduling, process migration, process creation, message passing and shared memory and process synchronisation. Each of these facilities are vital for the automated execution of parallel applications. This section presents the services required by the tool and how they are implemented.

The various characteristics of the execution environment services required by
the tool and how they are implemented are presented within this section. The structure of the GENESIS execution environment is shown in Section 4.7.1. This is followed by a brief description of the resource discovery, global scheduling and migration management services that are used indirectly by the tool within Section 4.7.2. The process creation, message passing and shared memory, and process synchronisation services which are explicitly invoked by the tool are presented in Sections 4.7.3, 4.7.4 and 4.7.5.

4.7.1 Structure of GENESIS

The chosen execution environment is called GENESIS [GOSC01]. It has been selected as it satisfies the execution requirements of this research, presented in Chapter 3. It also supports coarse grained parallel applications which fulfils the logical design with regards to granularity of the generated parallel applications.

GENESIS is designed to support parallel processing from the beginning. It consists of a microkernel (providing minimal services) and kernel servers (following the client server model). The microkernel provides the minimal core functionality required to support the kernel servers and other processes. The kernel servers provide services that support the higher level processes. Each kernel server is responsible for the one area of operation. If other areas are required, they are accesses via another kernel server. For example, when the IPC (InterProcess Communication) manager requires remote communications the Network Manager is given the task of providing that service.

The architecture of GENESIS is presented in Figure 4.13. The particular servers that provide the services the tool requires are highlighted (large dashed line). The servers that cooperate with the parallel application belong to the GENESIS parallelism management system [HOBB00]. The parallelism management system is responsible for the coordination of processes that constitute a parallel application. The management also applies to the efficient execution of parallel applications as a whole, in other words, all aspects of parallel application execution.
GENESIS provides two levels of support for parallel processing via microkernel and kernel servers and a parallelism management system. The support is in the form of services that are all transparent at the parallel processing level. The transparency is achieved by using well designed IPC mechanisms encased within primitives that have concise and consistent parameters. The parameters are used to communicate with the parallelism management system and use standard data structures which are used throughout GENESIS. The primitives are used by the servers of GENESIS to communicate with other servers to provide the required services and by the parallel processes to access the parallelism management system.
4.7.2 Resource Discovery, Global Scheduling and Migration Management

Resource discovery is required to gather information about the state of the execution environment and provide it to which ever service of the execution environment requires it. The global scheduler can use the information to assist in deciding about where to map a process and when to migrate processes (using the migration management service) as part of load balancing, for example. For the programmer, this means that there is no need to map processes manually, this also applies for automated generation of parallel applications.

Within GENESIS, there are two servers that provide the above mentioned services: the Resource Discovery Manager and the Global Scheduler [GOSC01]. The tool does not need to invoke these two servers directly, instead they are indirectly invoked through the process creation and termination operations.

The Resource Discovery Manager routinely gathers information about the state of the execution environment. When a request to create a process is made, the Global Scheduler uses the information provided by the Resource Discovery Manager to decide on which processor (or cluster computer) to map the process to.

Once parallel processes of the program are executing, the Global Scheduler balances the load of each processor. The load information provided by the Resource Discovery Manager allows the Global Scheduler to issue a request to a migration management service. In the chosen execution environment the migration management service is called the Migration Manager [DePA98]. The Migration Manager coordinates transferring a process from one location to another. Balancing processor load manually is a time consuming task, and is not possible in an automated approach without the services described in this section. There are no specific primitives used to invoke these services directly, they are employed indirectly. Therefore, the programmer automatically receives the benefits provided.

4.7.3 Process Creation

The parallel application generated by the parallelising compiler consists of
many parallel sections that are derived from the parallel units identified during the parallel analysis phases. The parallel sections are established by way of process creation. When a process creation primitive is invoked, the execution environment has two main tasks that need to be carried out. The two tasks are: allocating local system resources (memory, communication ports, and process control information) and mapping the process to a processor.

The server that coordinates the two tasks mentioned above within the chosen execution environment is called the Execution Manager [HOBB98]. Unlike the Resource Discovery Manager and Global Scheduler (see Section 4.7.2), this server is invoked directly by the parallel application. When a request is made for one or more processes, the Execution Manager contacts the Global Scheduler for a location or list of locations to map the processes to. The Execution Manager then contacts counterparts on these computer locations. Each EM then coordinates the allocation of the required local resources.

The information the tool specifies to invoke the process creation service is the name of the program image created by the tool and the number of processes. These pieces of information are specified during Phase 6 of the parallelising compiler. The name of the program image depends on the part it performs within the parallel application. The parallel application (or parent) is given the name of the original program. So, if the program is called fibonacci.p, then the parent is called fibonacci. The parallel programs within the parallel application are named with unique names for the parallel application. The unique names consist of the application name and a number (e.g. fibonacci_19750120_01). This is elaborated further in Section 4.8.1.

The number of processes is determined by the type of parallelisation section being created. If the MasterWorker model of parallelism is being followed, the num_iterations field of the syntax tree structure (see Figure 4.1) is set to one. This number is used when inserting the creation primitive. The number used is one because there are several different parallel processes being executed at the same time. For the SPMD parallelism model a similar technique to the MasterWorker model is followed. In this case the number of iterations will be greater than or equal to two.
This number is greater than one because the same program image must be created more than once. All other information that is required to create processes is determined automatically via the servers of the execution environment.

Within GENESIS process creation is provided by the process creation primitive (to be presented in Section 4.8.1) and supported directly by the Execution Manager. Indirect support is provided through the Global Scheduler and the Resource Discovery Manager.

### 4.7.4 Message Passing and Shared Memory

The message passing and shared memory facilities are provided by GENESIS to support the communication paradigms required by our tool. Message passing is required in order to send information to processes from the parent process and vice-versa. Almost all cluster-based execution environments provide such a service, and the chosen execution environment is no different. Shared memory is required to satisfy the higher level paradigm that the tool also utilises.

GENESIS provides support for message passing through its microkernel, IPC and Network managers (see Figure 4.13). The microkernel is responsible for any local messages and the IPC and Network managers handle any remote messages (incoming and/or outgoing).

The same message buffer can be used for a message that is destined for local or remote processes. The name of the destination process is non location specific also, which provides a transport interface that the programmer can use. When the destination of a message is remote, the network manager provides a reliable remote communications mechanism that ensures that the message reaches the destination correctly.

A process receiving messages is *blocked* as part of the implementation of the message passing receive message passing primitive. This blocking is carried out by the associated servers within a computer of the GENESIS cluster. This characteristic can be used deliberately to synchronise processes, especially when using the
The shared memory paradigm is provided in GENESIS by the DSM Manager (see Figure 4.13). The DSM Manager is a higher level paradigm than message passing. The message passing primitives are used by the DSM to carry out the necessary initialisation of a shared memory region and transmit updated information about the shared memory region between computers of the cluster. Synchronisation is also supported by the DSM Manager which is presented in Section 4.7.5 below.

A shared memory region allows the parent and child processes to share the same memory and therefore effectively pass information between each other. A shared memory region is identified during the low level code generation phase. A data structure (which is the shared element) is required to group identifiers together. The identifiers can be variables or complex data structures and information related to them are retrieved from the syntax tree and symbol table. The tool identifies and creates these data structures during the low level code generation phase.

To create a shared memory region as described above, a single primitive is provided by the execution environment (detailed in Section 4.8.3). The primitive is to be used within the parent process to carry out two operations, create \( n \) processes, and attach each process to a shared memory region. Therefore, the information used by the execution environment consists of a process image and the number of instances to create. The memory sharing creation and updating are handled internally within the execution environment.

GENESIS provides services to facilitate the Message Passing and Shared Memory paradigms with minimum required information. The implementation of the GENESIS servers responsible for handling these paradigms satisfy the logical design.

**4.7.5 Process Synchronisation**

A synchronisation service is required by the tool to enable a parallel application to stop executing in parallel and allow a sequential section to execute. The tool identifies these points within the parallel application and is able to structure the
Chapter 4 - Implementation of the automated parallelisation creation and execution tool

parallel application so that the appropriate execution environment services can be invoked.

There are two synchronisation types used by the tool. Firstly, via the parent waiting for the child parallel process(es) to complete processing. Secondly, all processes synchronise together before continuing processing.

For the first type of synchronisation the tool has to insert the appropriate primitive that requests the Process Manager of GENESIS (see Figure 4.13) to respond when any child process or processes have exited. The parent process will be blocked in a waiting state until the responses have been received. This is similar to the blocking for the message passing receive primitive, see Section 4.7.4.

For the second type of synchronisation the tool inserts a primitive that contacts the DSM Manager (see Figure 4.13) and waits for a response. All processes using this type of synchronisation must share a common variable (via shared memory) so that the DSM Manager is able to tell when all processes have invoked the primitive. Therefore, the second synchronisation method can only be used if a parallel section within a parallel application is using shared memory.

A subtle variation on this second type of synchronisation is a critical section of code. When a critical section is identified, a primitive requesting exclusive access to a shared variable (commonly called a semaphore) is invoked. This primitive contacts the DSM Manager which coordinates all the processes that are requesting access to the semaphore. The DSM Manager ensures that deadlock does not occur between processes therefore relieving the programmer from having to provide extra code to cater for the deadlock situation.

Regardless of the synchronisation method, the tool is responsible for deciding when and where to utilise synchronisation. The GENESIS execution environment servers follow the instructions of the generated parallel application through providing their service automatically and transparently.
4.8 GENESIS Execution Environment Primitives

The main focus of this research is that the programmer must not be involved in either the parallelisation process or execution process.

The objective of Phase 6 of the parallelising compiler is to insert the execution environment primitives within the parallel application. These primitives are required to link the parallel application to the execution environment. This allows the parallel application to gain access to the services required for parallel execution. The primitives are inserted when the appropriate conditions are met.

This section presents the primitives that are supplied to gain access to the services identified in Section 4.7. The GENESIS specific primitives which satisfy the primitives specified in the logical design are `process_create()`, `process_wait()`, `dsm_barrier()`, `lock()` and `unlock()`. Each primitive is presented as part of the GENESIS servers that provide the services which are invoked by the primitives.

4.8.1 Execution Manager Primitive

To access the process creation service of the Execution Manager the primitive `process_create()` is provided [HOBB96]. This primitive requires a set of parameters which is shown in Figure 4.14. These parameters are:

- proc_name – The name of the child process to be created.
- num_procs – The number of child processes to be created.
- child_psns – The array of process sname’s of the created children.
- child_uports – The array of unique port sname’s of the created children.
- argv – The list of arguments to be passed to the children.
- envp – The list of environment arguments to be passed to the children.

The list of parameters for this primitive is divided into two sections. The first section is used to pass information to the Execution Manager. The second is

---

1. SNAME is the GENESIS system name, a generic type to identify all processes, communication ports, etc.
Chapter 4 - Implementation of the automated parallelisation creation and execution tool

The parameters proc_name, num_procs, argv and envp contain the information to be passed to the Execution Manager. The child_psns and child_uports parameters are populated by the Execution Manager and can be used by the calling process once the primitive has returned.

This process_create() primitive is inserted into the parent application during the low level code generation. At this point the program code for the parallel child program is stored in a separate file. This separate file constitutes the proc_name parameter of the process_create() primitive. The generation of

```c
int process_create (  
    char *proc_name,  
    int num_procs,  
    SNAME child_psns[],  
    SNAME child_uports[],  
    char *argv[],  
    char *envp[]  
);  
```

Figure 4.14 Process Creation Primitive

```c
/* At the beginning of the parallelising compiler */
...
epoc = get current time in seconds();
slave_process_number = 0
...
/* During low level code generation. */
child_process_name = generate_unique_filename();
...
/*
 ** This function will return a unique slave
 ** process filename
 ** e.g. process_1009803600_01.exe
 */
string generate_unique_filename()
{
    increment slave_process_number;
    filename = "process_" + epoc + ":" +
               slave_process_number + ":.exe";
    return ( filename );
}
```

Figure 4.15 Algorithm used to create unique parallel process name.
the separate file is carried out during Phase 6, the low level code generation phase (see Section 4.9). The name of this file is created via a time based number generator that creates a unique name for the whole application and other applications. The algorithm used to create this file name is presented in Figure 4.15. The code for the parallel program is created from the information within the syntax tree.

The `num_procs` parameter is used for both the SPMD and MasterWorker models. When the MasterWorker model has been followed, several separate parallel processes are executed together and therefore only one copy of each process is to be created. The `num_procs` parameter is therefore set to one when the MasterWorker model is being followed, an example is shown in Figure 4.16.

For processes created as part of a loop that has been parallelised, any number of processes can be specified. The exact number used is taken from the syntax tree. The value is determined from analysis on the loop indices. In some cases a constant is used and therefore the number is an absolute constant value. When identifiers containing dynamic values are used, the identifiers are then represented as symbols. The symbols are translated to low level code after linking the primitives. This means that the `num_procs` parameter for the `process_create()` primitive is a variable and not a constant number. This is a minor point, but an important one as it shows how non constant values for loop indices can be used for parallelisation.

```c
/* A MasterWorker section sample */
if ( process_create("process_1009803600_01.exe", 1,
    &cpsn, &cuport, &cargv, &cenvp ) == -1 )
    error("process_create");
if ( process_create("process_1009803600_02.exe", 1,
    &cpsn, &cuport, &cargv, &cenvp ) == -1 )
    error("process_create");
... /* more process_create()’s */
if ( process_create("process_1009803600_03.exe", 1,
    &cpsn, &cuport, &cargv, &cenvp ) == -1 )
    error("process_create");
...  
```

Figure 4.16 MasterWorker model implementation
4.8.2 Process Manager Primitive

The primitive `process_wait()` (see Figure 4.17) is provided to access the synchronisation service the Process Manager provides [DePa94]. The parameters for this primitive are only provided for the Process Manager to populate. They are as follows:

- status – The value returned from an exited child process.
- child – The process sname of an exited child process.

```c
void process_wait(
    int *status,
    SNAME *child
);
```

**Figure 4.17 Wait primitive**

Synchronisation is invoked through this primitive by waiting for specific parallel children to complete execution. The specific children are noted by the `child` parameter. A loop of this primitive enclosing a check of the `child` parameter ensures that the anticipated children processes have exited.

4.8.3 DSM Manager Primitives

When a shared memory is used for communication between two or more processes the `start_dsm()` primitive is provided [SILC99]. The parameters of this primitive are presented in Figure 4.18.

```c
int start_dsm(
    char *client,
    SNAME *sp_name,
    int consistency,
    int size,
    int *num_procs,
    SNAME *sem,
    SNAME *barrier,
    int numsems,
    int numbarriers
);
```

**Figure 4.18 Shared memory creation primitive**
Each parameter represents the following:

- **client** – The name of the child process to create.
- **sp_name** – The space name (memory handle) of the shared memory space.
- **consistency** – The type of memory consistency model to use (RELEASE or SEQUENTIAL.)
- **size** – Size of the shared memory region.
- **num_procs** – Total number of processes attached to the shared memory region.
- **sem** – The array of semaphores to be used through the processes.
- **barrier** – The array of barriers to be used through the processes.
- **numsems** – The number of semaphores.
- **numbarriers** – The number of barriers.
- **RETURN VALUE** – Base address of the shared memory region.

As with the parameters of the Execution Environment primitives this primitive also has two sections. The parameters **client**, **consistency**, **size**, **numsems** and **numbarriers** are all used to pass information to the DSM Manager. The parameters **sp_name**, **sem** and **barrier** are all populated by the DSM Manager. The **num_procs** parameter is both passed to and populated by the DSM Manager.

The **client** parameter is specified in the same manner as the **proc_name** parameter of the **process_create()** primitive. The **consistency** parameter is used to specify the consistency model of the shared memory region. In this research the **release** model is used as it is faster than **sequential** [SILC98b]. The **size** parameter is specified by the parallelising compiler as the size required for all the identifiers located and used within the shared memory region. This is determined in Phase 6 when the parallel program is generated. The **num_procs** parameter is set to a value similar to the **process_create()** primitive. In this case it is always a value at least two, so that a shared memory region can be shared.

Through using the **start_dsm()** primitive the tool is able to create processes that share a memory region. The underlying initialisation of the shared memory region is transparent to the programmer when using this primitive. When the parallel application uses the **start_dsm()** primitive, the same transparency is
observed. The transparency of this primitive is vital, as the underlying initialisation is quite complex and not something a parallel application programmer should be performing.

To access the semaphore and synchronisation services of the DSM Manager the primitives `lock()`, `unlock()` and `dsm_barrier()` are provided [SILC98a]. The semaphore primitives `lock()` and `unlock()`, have one parameter called `lock` that specifies the particular semaphore the critical section is referring to. The `lock()` primitive pauses the process until it is able to obtain a lock for the requested semaphore. When it is able to obtain the lock it then enters the critical section of code. After the critical section has completed processing an `unlock()` primitive is used to release the lock, which the process had previously obtained. At that point, the lock is free for another process to use. These two primitives are presented in Figure 4.19.

```
int lock(
    SNAME lock
);

int unlock(
    SNAME lock
);
```

**Figure 4.19** Shared memory lock and unlock primitives

The parallelising compiler uses these two primitives in two different ways, explicitly and implicitly. The explicit usage comes from the information within the syntax tree. The information specifies when a semaphore is required and the low level code generation phase uses this information to insert the `lock()` and `unlock()` primitives. The implicit usage is derived internally within the low level code generation phase as determined when creating parallel programs with shared memory. The `lock` parameter used for both primitives is generated internally within the low level code generation phase. The parameter is required to specify which semaphore to wait for or release.

Each parallel program that is involved in synchronisation must insert the synchronisation primitive called `dsm_barrier()` (see Figure 4.20) [SILC98a]. The parameter `barrier` is automatically generated in a similar manner to the `lock` parameter of the semaphore primitives. All the parallel programs involved in the
synchronisation must belong to the group of programs that share a common memory region whilst executing. When invoked, the primitive contacts the DSM Manager with the specified barrier, the process then pauses. Once all processes that are sharing the same memory region have invoked the dsm_barrier() primitive they are all released from their paused state, to continue processing.

\[
\text{int dsm\_barrier(}\hspace{1cm} \\
\phantom{\text{int dsm\_barrier(}} \text{SNAME barrier}\hspace{1cm} \\
\text{)};\]

Figure 4.20 Shared memory synchronisation primitive

The lock(), unlock() and barrier() primitives allow the parallelising compiler to place the appropriate semaphore and synchronisation conditions into the generated parallel application. The underlying operations are carried out transparently by the DSM Manager.

4.9 Implementation of Phase 6 - Low Level Code Generation

This phase is responsible for generating the low level code that constitutes the parallelised application. This requires use of the syntax and symbol table structures that have been passed between every phase. The syntax tree at this phase contains several different pieces of information, including flags that identify parallel sequences and synchronisation points. The start_sequence and end_sequence flags signify the parallel sequences and the sync_point flag signifies the synchronisation points.

The language used for the low level code generation is the C language. Using the C language enables the execution environment primitives (presented in Section 4.8) that are supplied and used to be easily linked into the parallel application generated by the tool. Once the parallel application and associated parallel programs have been generated as files of C language code, they are all converted to machine specific code so that the whole parallel application can be executed on the destination execution environment, GENESIS. This machine code generation is similar to any
compiler, but the input to this conversion (i.e. the C language code) is different due to the parallelisation performed in prior phases. The conversion to machine code is performed by the GNU C Compiler (GCC) [FSF99].

This section addresses the methods used to generate the low level code from the syntax tree and symbol table information that has been passed through all phases. The process used to generate parallel applications with the Message Passing paradigm being used, is presented in Section 4.9.1. How the Shared Memory paradigm is used when generating a parallel application is presented in Section 4.9.2. The synchronisation and information dissemination methods used for Message Passing and Shared Memory are presented in these two sections also.

4.9.1 Generating Parallel Applications using Message Passing

The majority of generated code is the same for both Message Passing and Shared Memory, but there are some slight differences that need to occur. When the programmer specified that Message Passing is to be used, the techniques are as follows.

The method used to generate the low level code starts by traversing the syntax tree. The operator of each element/node of the syntax tree is consulted to determine what low level code to generate (i.e. a translation). When operators such as assignment, loops or function/procedure calls are reached, further information from the modified syntax tree needs to be examined.

- For identifiers, the `sym_tab_ptr` (see Figure 4.1) element of the syntax tree structure is used to get the details of the identifier from the symbol table.
- For loop operators, the number of iterations need to be determined.

The number of iterations can be static or dynamic, an example of each is presented in Figure 4.21. The static value is calculated via constants which is performed within the parallel compiler, this can be seen on the left of the figure. The dynamic value is determined via placing the actual identifiers in the low level code as an equation; this can be seen on the right of the figure.
Chapter 4 - Implementation of the automated parallelisation creation and execution tool

Generation of the low level code results in numerous statements written to output files. The output file created for the main (or parent) part of the program is given the same name as the supplied sequential application name. Parallel sections of the application that are identified are placed in separate files with a unique name. The unique name consists of the application name and a time based unique number.

To generate the low level code the syntax tree is traversed. As each node is passed the contents of the node (i.e. the syntax tree structure, see Figure 4.1) are checked. When the start_sequence flag is set, the code following in the syntax tree up to the node with the end_sequence set is placed in a separate file. At this point in the code for the parent process, the process_create() primitive is inserted. This is the primitive needed when the Message Passing paradigm is being followed. The first parameter for this primitive is the name of the separate file that has been generated. This file is to become the parallel child process when it is instantiated.

The algorithm that is used to insert the execution environment primitives and create the parallel processes are presented in Figures 4.22 and 4.25 respectively. The first section of the algorithm (Figure 4.22a) is where the syntax tree is traversed and each node is consulted via the operator element. In the example shown in Figure 4.22, the case of the FOR operator is presented.
Chapter 4 - Implementation of the automated parallelisation creation and execution tool

```
traverse the syntax tree
switch on the operator
... case FOR:
    /* If the criteria/is satisfied then
    ** a primitive will be inserted.
    ** i.e. The start_sequence field is set. */
    /* This function generates the appropriate low
    ** level code to calculate the number of children
    ** If a constant value could not be found. */
    num_children = calc_num_children(
        num_iterations);
    printGenesisPrimitive( CREATE, output_file,
        child_info, num_children, syntax_tree_ptr );
    /* Generate code for parallel process
    ** (see Figure 4.23) */
    gen_child_ident_info_code( output_file,
        num_children, syntax_tree_ptr );
    /* Synchronise */
    printGenesisPrimitive( WAIT, output_file, NULL,
        number_of_children, syntax_tree_ptr );
    /* Generate code to receive identifier
    ** information from the child process */
    break;
...
```

Figure 4.22 Insertion of execution environment primitives pseudocode example

```
/*
** primitive_name == the name of the primitive
** file_pointer == internal file pointer to save the generated
** code child_information == various pieces of information
** about a child process (can be NULL)
** num_children == number of child processes to instantiate
** (can be -1 if number was not able to be determined
** syntax_tree == internal pointer to the syntax tree, in
** particular where the tool is currently pointing to
** */
printGenesisPrimitive( primitive_name, file_pointer,
    child_information, num_children, syntax_tree )
{
    switch on the primitive_name
    case CREATE:
        string = "if( proc_create( " +
        child_information->path + ":");
        if ( num_children == -1 )
        |
        string = string + "num_children";
        else
        |
        string = string + num_children;
        ( string = string + ")" ) < 0 )";
        print_c_output( file_pointer, string );
        print_c_output( file_pointer,
            handle_error_string );
    ...
```

b) Section 2
Chapter 4 - Implementation of the automated parallelisation creation and execution tool

The calc_num_children() function returns an integer which represents the number of child processes to instantiate. The num_iterations parameter is a pointer to the syntax tree used to determine the number of child processes. If the loop indices are static constants, then the function will return a positive integer, if they are found to be dynamic, then negative one is returned and an extra equation is placed into the generated low level code before the process_create() primitive.

The low level code for the GENESIS primitives are generated when the print_genesis_primitive() function is called. This function is defined in Figure 4.22b where the CREATE example is shown. The num_children parameter is inspected to define the low level code to generate, highlighted within the dashed box (lines 16–19).

After the process creation primitive is inserted, the parallel code that constitutes the parallel process (or child) is then generated. The necessary steps for generating the parallel program code are shown in Figure 4.23. To begin with the unique file name for the child program is opened (line 3). An internal list of identifiers is generated next (lines 7–8). This list is a simplified form of the identifiers

```
1    /* This function creates a unique file name over the whole
2    ** parallel application */
3    child_output_file = open_unique_outputfile();
4
5    /* Place the necessary global variables for the execution
6    ** environment in child_output_file */
7    traverse syntax tree until the end_sequence flag is set
8    adding identifiers used (both read and set) to a list
9
10   if ( message passing selected )
11      generate code for receiving initial state of variables
12
13   /* Similar traversal/switch mechanism as shown in Figure 4.22.
14   generate code for the child placing it in child_output_file
15
16   if ( message passing selected )
17      generate code for sending written identifiers back to
18      the parent process
19
19   generate code for the appropriate execution environment
20     primitive to exit the child process
21   close child_output_file
```

Figure 4.23 Parallel process code generation pseudocode
used within the parallel child program that is being generated. The list consists of
pointers to the identifiers within the syntax tree and is used to generate the low level
code that is required for Message Passing (lines 10–11), in particular setting up the
memory buffer. This list is used to set up the message buffer to send back the
computation results also. The main low level code generation for the body of the child
program is performed next (line 14). After the body of the child program has been
generated the low level code to send the results back to the parent is generated (lines
16–17). This is followed by the low level code to exit the child (line 19) and then
close child program file (line 21).

**Synchronisation and Information Dissemination**

At the stage where synchronisation of a parallel application is to occur, there
may be several different programs generated which will all execute at the same time,
or the same program that is created \( n \) times and executed at the same time. In either
case the syntax tree is traversed until a node is found with the `sync_point` flag set.
The appropriate synchronisation primitive is inserted at this point.

For the Message Passing paradigm the `process_wait()` primitive is
inserted within the parent to synchronise all the child processes. Within the generated
child processes, the results of computations need to be sent to the parent for
coordination as part of the synchronisation process. All the values of the identifiers
that have been modified within the parallel process are packed into a buffer and sent
to the parent process. To generate the low level code required to pack the values into a
message buffer, the list of identifiers created at the beginning of the generated parallel
process is used. After this code has been generated the primitive used to synchronise
the processes, `process_wait()`, is then inserted. This can be seen in Figure 4.22a
when the `print_genesis_primitive()` function is called with the `WAIT`
parameter. By using the one list of identifiers throughout the parallel process
generation, functions within the parallelising compiler can be reused.

The algorithm used to generate the low level code to pack the message buffer
and send it to the parent is shown in Figure 4.24. Part of the message buffer packaging
is specifying the size of the individual elements, shown at line 9.
When a loop is split so that each iteration of the loop executes as a separate process (i.e. following the SPMD model), the value of the index of the loop must be passed to each parallel process. This index is the value that each iteration of the loop is based upon within the loop condition bounds. For example, a matrix multiplication where all the entries of a matrix are being used. To do this one or more iterators (or

```c
void gen_child_ident_info_code( output_file, num_children, syntax_tree_ptr )
{
    for ( ident = identifiers_list; ident != NULL;
     ident = ident->next )
        string = "msg.buffer = " + ident.name + ";
        string += "msg.size = " +
            get_size_of_identifier(ident) + ";";
        print_c_output( output_file, string );
    string = "index = " + get_left_idx_val(syntax_tree_ptr) + ";";
    print_c_output( output_file, string );
    string = "for ( cnt = 1; cnt < ";
    if ( num_children == -1 )
        string += "num_children";
    else
        string += num_children;
    string += "; cnt++, index++ ) { ";
    print_c_output( output_file, string );
    string = "msg.buffer = index;"
    string += "msg.size = sizeof(index);"
    print_c_output( output_file, string );
    /* send message to children */
    string = "send( child_port, parent_port, msg, results );"
    print_c_output( output_file, string );
}
```

Figure 4.25 Algorithm for sending loop index information
indices) are required. The number depends on the dimensions of the matrix. When the parallel processes are created the value of these indices are required for each parallel process instance. These values are sent along with the identifier values that are sent to each parallel process when instantiated. The `gen_child_ident_info_code()` function used for this is shown in Figure 4.25. This function is called from the main syntax tree traversal code shown in Figure 4.22.

Appendices C, D and E contain an example of a parent, a child and miscellaneous generated files respectively, represented as low level code using the message passing paradigm. This is the appearance of the parallel application before it is compiled to machine code and executed on the execution environment. The appropriate synchronisation and information dissemination primitives within the parallel application are all shown.

### 4.9.2 Generating Parallel Applications using Shared Memory

The same syntax tree traversal technique as used with Message Passing, is used when the Shared Memory paradigm is being used to generate the appropriate low level code. The list of identifiers created for Message Passing is also used for Shared Memory to initialise the shared memory region after the child process(es) have been created, but before computation begins.

Within the child parallel program the information for the identifiers is not extracted as for the message passing solution. Instead all references to these identifiers are passed through the shared memory space which is implemented by generating the appropriate code as shown in Figure 4.26.

The other differences that have been implemented for the generation of a parallel application using Shared Memory are related to synchronisation and the dissemination of information. These differences are described below.

**Synchronisation and Information Dissemination**

Synchronisation with the Shared Memory paradigm utilises three primitives,
Chapter 4 - Implementation of the automated parallelisation creation and execution tool

The barrier() primitive is used to synchronise parallel sections of the parallel application. The lock() and unlock() primitives are used to ensure that critical sections of the application are synchronised so that corruption does not occur.

For Shared Memory (as with Message Passing) the information from the child parallel programs need to be extracted at their completion. When executing, the child processes need to invoke a synchronisation primitive that informs the parent process that the shared memory region has possibly changed. The barrier() primitive is used to do this. This is inserted into the child program and the parent program as shown in Figure 4.27 using the print_genesis_primitive() function defined in Figure 4.22b. The code to extract the information uses the same structure mechanism that is shown in Figure 4.26. The switch statement within the extract_information() function determines which method to copy from the shared memory region.

When a critical section of code has been detected during the parallel analysis phases, the semaphore element of the syntax tree structure is set. This means the synchronisation primitives that “protect” the critical section of code need to be inserted. To do this the signal() and wait() primitives from the execution environment, lock() and unlock() respectively, are inserted into the generated application. The parameter to these primitives is a semaphore which is shared between all parallel processes when shared memory is being used. The code used to insert these two primitives is shown in Figure 4.28.
Chapter 4 - Implementation of the automated parallelisation creation and execution tool

In a similar manner to Message Passing, when a loop has been split according to the SPMD model, the index values must be passed to each child parallel process. However, if the index identifier name was added to the shared memory structure and set to a value, then every parallel child process would receive that same value and the whole application would be void. To solve this issue the index values are sent via the Message Passing solution presented in Section 4.9.1.

Appendices F, G and H contain an example of a parent, a child and miscellaneous generated files respectively, represented as low level code using the shared memory paradigm. As with the message passing examples, the representation of these files is how the parallel application appears before compilation to machine

```c
/* Traversing the syntax tree */
...
print_genesis_primitive( START_DSM, output_file, num_children, 
syntax_tree_ptr );
print_genesis_primitive( BARRIER, output_file, 0, NULL );
/* Extract information from shared memory region */
exttract_information( output_file );
...
void extract_information( output_file )
{
    while ( identifiers ) {
        switch ( identifier_type ) {
            case integer:
                string = identifier + " = sm_" + childno + "_" + 
                         structure_number + "." + identifier + ";";
                break;
            case array:
                string = "memcpy(" + identifier + ", ", " + "sm_" + 
                        childno + "." + structure_number + ");";
                break;
            ...
        }
        print_c_output( file_pointer, string );
    }
}

/* Other low level code for the child generated */
...
print_genesis_primitive( BARRIER, child_output_file, 0, NULL );
print_genesis_primitive( EXIT, child_output_file, 0, NULL );
```

Figure 4.27 Algorithm to insert barrier() synchronisation primitive.

In a similar manner to Message Passing, when a loop has been split according to the SPMD model, the index values must be passed to each child parallel process. However, if the index identifier name was added to the shared memory structure and set to a value, then every parallel child process would receive that same value and the whole application would be void. To solve this issue the index values are sent via the Message Passing solution presented in Section 4.9.1.

Appendices F, G and H contain an example of a parent, a child and miscellaneous generated files respectively, represented as low level code using the shared memory paradigm. As with the message passing examples, the representation of these files is how the parallel application appears before compilation to machine
Chapter 4 - Implementation of the automated parallelisation creation and execution tool

4.10 Parallel Application Execution

Once the primitives have been inserted and the low level code has been generated, the parallel application is complete. The only task left is to execute the parallel application. This is carried out by a batch script that the tool creates and executes itself. For the script to work, there needs to be a Makefile generated for the Unix “make” utility, an example Makefile is shown in Appendix A. The algorithm used to create the batch script is shown in Figure 4.29 and is created after the low level code of the parallel application has been generated. The batch script tells the execution environment where to find the parent process of the parallel application, and then instantiate that process. The C library call “system” is used to invoke the script. An example batch script is shown in Appendix B. The script allows the tool to automatically execute the parallel application and hence the programmer is not

traverse syntax tree
    if ( semaphore operator set to signal )
        print_genesis_primitive( SIGNAL, output_file, NULL, 0, syntax_tree_ptr );
    elseif ( semaphore operator set to wait )
        print_genesis_primitive( WAIT, output_file, NULL, 0, syntax_tree_ptr );
    ...

/* print_genesis_primitive() continued from Figure 4.22 */
...
  /* In both SIGNAL and WAIT, ident_lock is a global
     ** variable that is registered with the shared memory
     ** manager within the execution environment. */
  case SIGNAL:
    ident = syntax_tree->identifier;
    printf( file_pointer, "lock(ident_lock);" );
    break;
  case WAIT:
    ident = syntax_tree->identifier;
    printf( file_pointer, "unlock(ident_lock);" );
    break;
  ...

Figure 4.28 Semaphore insertion code and execution on the execution environment. The process creation and synchronisation primitives are all shown. The code used to access the shared memory region is shown also.
involved in any part of the execution of the parallel application.

```
if ( (ofp = fopen(batch_script_name, "w")) != NULL ) {
  perror("fopen")
}
fprintf(ofp, "#!/bin/bash\n");
fprintf(ofp, "# Execution batch script for the "parallel_application_name);  
fprintf(ofp, "cd %s\n", output_directory);  
fprintf(ofp, "make -f Makefile.%s\n", parallel_application_name);  
fprintf(ofp, "if [ $? = -eq 0 ]; then\n");  
fprintf(ofp, "\tscc -auto_kbuild rhodos-3a -send_ctrlc  
    -def %s_defaults_file\n", parallel_application_name);  
fprintf(ofp, "fi\n");  
close(ofp);
```

Figure 4.29 Code to create execution batch script and execute the parallel application

To conclude, the parallelising compiler examines a source program written in the procedural language Pascal. It then carries out parallel analysis utilising internally constructed data structures. A parallel application is generated from this parallel analysis that contains the appropriate information for it to be executed on the chosen execution environment. The information is passed to the execution environment in the form of primitives. The execution environment is instructed by the tool to instantiate the parallel application which then follows the instructions specified by the inserted primitives.

### 4.11 Summary

In this chapter the implementation of the automated parallel application creation and execution tool was presented. The implementation of this tool satisfies the research aims of automatic creation of a parallel application and linking that application to an execution environment.

The tool comprises two components, a parallelising compiler and the GENESIS execution environment, that are linked together so that a sequential application is parallelised and executed without any extra input from the programmer. The parallelising compiler carries out the necessary analysis required to create the
parallel application.

Each phase of the parallelising compiler plays an important role in the overall parallelisation of the tool. The first phase creates the two data structures, a syntax tree and symbol table. These two data structures are used throughout all subsequent phases of the compilation process. The following three phases perform the necessary parallel analysis of the tool. The first of these three phases (Phase 2) produces a parallel symbol table that is used to assist the identification of the units of parallelism and sequences. The second of the three parallelism phases (Phase 3) uses the parallel symbol table and the syntax tree to identify sequences in the previously identified units of parallelism. Appropriate synchronisation between the groups of sequences is identified during the third of the three parallelism analysis phases (Phase 4). As well as the parallel symbol table, used in Phases 2 and 3, all of the parallel analysis phases produce results that are stored within the syntax tree. The outcome of the first five phases of the parallelising compiler is a syntax tree data structure that contains updated elements that provide the sixth phase with the information it needs to generate low level code. The information consists of when a section of the syntax tree should be executed in parallel, starts and ends, and when those parallel segments are to be synchronised. This information provides an overall picture of the (now) parallel application.

The primitives used in the parallel application have been selected to satisfy the requirements of parallel processing within this research. The GENESIS execution environment provides many primitives for parallel processing. The specific primitives presented in this chapter have been chosen as they satisfy the requirements specified in the logical design. They are efficient in terms of the concise amount of information they require and how they invoke services of the GENESIS execution environment. The primitives are used during the sixth and final phase of the parallelising compiler, when the low level code is generated. They are inserted into the low level code with parameters derived from the information contained within the syntax tree.

The naming of each process is important at this point as there can be many different processes executing in parallel at the same time. To make sure that there are no conflicts, the compiler selects unique names based on the application name and the
current time. This results in each parallel process of the parallel application being unique for that application.

Once the parallel application has been created, the programmer is not required to explicitly setup the execution environment and instantiate the application. The tool initialises the GENESIS cluster itself via a batch script. At this point GENESIS takes over and executes the application on behalf of the programmer. The mappings of processes to computers and other process coordination is carried out automatically by the servers of GENESIS.

The achievements of the implementation of the logical design contribute to a cohesive tool that carries out the tasks of creation and execution of a parallel application automatically. The modular multi-phase approach to the parallelising compiler facilitated clear development of the tool. Also, the selection of GENESIS meant that linking the parallel application to an execution environment was simplified. Many of the traditionally manually intensive tasks of parallel execution have been removed by the GENESIS servers.
Chapter 5  
Assessment of the Automated Parallel Application Creation and Execution Tool

5.1 Introduction

The automated parallel application creation and execution tool should relieve programmers from many operating system oriented programming activities and improve the performance of execution of parallelised applications. Thus, to address these two major issues and also the feasibility and ease of use of the tool certain testing and analysis is required. The aim of this chapter is to address these aspects of the tool, through qualitative and quantitative assessment.

A “proof-of-concept” is a very important element of experimental computer science. Thus, in Chapter 4 the implementation aspects of the feasibility of the tool are addressed. The testing aspect, entails the ability of the tool to actually produce and execute a parallel application in a real life situation automatically; and also assess the amount of input the programmer is required to provide. This is the qualitative assessment component which enables us to determine if the tool is assisting the programmer as it is required (i.e. proof-of-usability).

The performance of a parallel application created by the tool developed as part of this research is characterised by the results produced when the parallel application is executed. This is the quantitative (or proof-of-performance) assessment component.

The tests reported on in this chapter cover two applications that the tool executes. The first application contains recursive and loop elements as well as routines that have parameters that are pass by reference and therefore dependent. The second application contains iterative loop elements. Both are used to test the ability of the tool to parallelise and execute the parallel applications that are generated.

To present the proof-of-usability and proof-of-performance assessment of the
Chapter 5 - Assessment of the Automated Parallel Application Creation and Execution Tool

This chapter is structured as follows. The criteria for the assessment of the tool are presented in Section 5.2. The characteristics of the testing environment, including the execution environment and tool are shown in Section 5.3. The objectives and scopes of the test programs are given in Section 5.4. Following this in Section 5.5, the results of the tests and findings are demonstrated. A summary of the assessment is presented in Section 5.7.

5.2 Criteria for the Assessment

Two types of testing are required to assess the tool: firstly, the qualitative assessment of the tool, and secondly, the quantitative assessment of the tool. The criteria for both of these need to be established before any tests are presented. In this section, these criteria are discussed.

5.2.1 Usability Criteria

The logical design and implementation of the automated parallel application creation and execution tool have shown that the programmer does not have to provide any input into the parallelisation process at all. A manual approach requires extraordinary programming skills that a traditional sequential programmer does not readily possess. To assess the feasibility of the tool from the programmers point of view the usability needs to be assessed. This is achieved by comparing the manual and automated techniques side by side.

The design of our tool is such that it does not require a new programming language for the programmer to be learnt. The programmer is required to write a sequential application using a procedural language (in this study we used the Pascal programming language). No extra directives are required to assist the tool to create a parallel application. These characteristics contribute to reducing the burden on the programmer.

The tool is designed to make all actions, from parallel application creation to execution, completely automatic. The information required to allow the execution environment to execute is as little as possible, as the execution environment has been
designated to provide services transparently. The parallel application developer therefore does not have to specify every execution environment oriented detail required to initialise and execute the parallel application.

When the parallel processes of a parallel application are created a location for their execution is required. Within the chosen execution environment the specific location is allocated automatically. The allocation (or mapping) is carried out via the parallelism management system of the execution environment. Once each process is created, their communication end points are also created automatically. These factors allow the programmer to concentrate on the original program solution.

The effects of message passing and shared memory on the generated parallel applications is also assessed. When using message passing, the information required to be passed between the parent and child processes can be carried out quite easily. However, programming using this approach is demanding and error prone. The individual pieces of information that are sent and received must be specified explicitly, and then passed to the appropriate entity. When using shared memory, the information passing used with message passing is simplified. Synchronisation is also simplified and extended with shared memory.

The results of this assessment aim to show that the tool relieves the programmer from many activities which are of the operating system nature and is very easy to use, thus satisfying the ease-of-use research aim. This vindicates the statement that this tool makes the programmers life easier, and the whole process of parallelisation and execution much cheaper.

5.2.2 Performance Criteria

The basis for the performance assessment of the tool is the speed-up\(^1\) provided by the automatically generated parallel application against the original sequential application. To assess the speed-up, a time quantum needs to be obtained at the start and end of the execution of the application. The difference in time represents the

\(^1\) The term speed-up is defined to be the ratio of the total time for execution of a sequential application versus the total time for execution of the parallel version of the application.
elapsed execution time. This elapsed time includes all computation and underlying parallel process management that occurs during the life of the parallel application. When the elapsed time of a sequential and parallel application are compared a speed-up (or down as the case may be) can be determined.

A positive speed-up represents the decrease in execution time for the parallel application (i.e. a time saving for the programmer and therefore the end user). A negative speed-up represents an increase in execution time for the parallel application.

Previous work examining the GENESIS execution environment services (process creation, duplication, migration and DSM), analysed only the specific services within existing parallel applications. The time to create a parallel application has never been included and is not for this research either as there is a human element for the original solution (sequential or parallel). The speed-up observations of this research (as described above) are used to provide a realistic value of the overall application execution time, from the beginning of the application creation to the end of execution. This does not include the parallel compilation time.

Therefore the performance criteria for the assessment is the speed-up of a parallel application created by the tool. This criterion contributes to satisfying the proof-of-performance research aim.

5.3 Testing Environment

The testing carried out was primarily to establish the usability and performance of the automated parallel application creation and execution tool. The environment used was a cluster based execution environment called GENESIS. The setup of the tool along with the environment is presented in this section.

5.3.1 The Tool

The tool has been implemented as a single application that is executed via a command line interface. The command line is used to execute the tool which includes
invoking GENESIS and providing it with a parallel application. There is currently no command line interface to GENESIS that provides tools such as gcc and make. Therefore an external UNIX (Solaris or Linux) operating system (which does provide these tools) is used to execute the tool. From the command line there is only one option to select whether to use message passing or shared memory, which reflects the communication paradigm used in the application. The only other piece of information required is the sequential program itself, which constitutes the input of the tool.

The sequential program is created by the programmer as a solution to a particular problem. The method used to create the program is the same as for any procedural program and consists of one text file containing all the code. The tools used to create this sequential program are a text editor or an integrated development environment (IDE) - if the programmer chose to.

The parallel application created by the tool contains C code with the inserted execution environment primitives, which is then converted to platform specific machine code. This is performed by the GCC compiler [FSF99], which has been setup as a cross compiler for the Sun3/50 (Motorola 68020) architecture. This allows the automated parallel creation and execution tool to be executed from the UNIX based interactive environment. The location of the UNIX front end in relation to the execution environment is shown in Figure 5.1.

The GENESIS execution environment is then initialised automatically via the SCC\(^1\) tool, which is used to interact with the cluster. Once GENESIS is initialised, the parallel application is executed.

### 5.3.2 The GENESIS Execution Environment Platform

The parallel applications produced by the automated parallel creation and execution tool were tested on a cluster of ten Sun3/50 computers connected via a 10Mbits/s shared ethernet network. The topology of this cluster is presented in Figure 5.1. Of these computers, one is a dedicated file server leaving the remaining nine computers for computation. All but two of the computers have 4 megabytes of

---

1. Serial Connection Client - a GUI to interact with all computers of the GENESIS cluster.
memory, the other two have memory expansion boards to provide a total of 8 megabytes of memory.

The speed of the processors is only 1.5 MIPS (million instructions per second) or 15.7 MHz. This does not adversely affect the testing environment as the relative speed (i.e. speed-up) is the focus of this research.

To gather the time information for testing, time stamps are inserted into the parallel application one at the start and again at the end of the execution. The times taken represent the elapsed time, which is used for the results assessment.

Each of the files generated by the tool are accessible to the cluster through the dedicated file server (see Figure 5.1). The file server within GENESIS communicates with an external file server that has direct access to a file system. The communication is via the native GENESIS communication protocol, RRDP [MCAV97b]. The external server sits on a UNIX operating system. The file server within GENESIS caches files from the external file system so that multiple accesses are delivered more quickly [HOBB95].

At this point it must be pointed out that there are obvious limitations imposed by GENESIS. The memory resource of each computer is quite small and therefore the

---

1. At the time of writing this thesis, a porting project to the PC platform is being conducted. This will provide greater flexibility for scalability and absolute performance testing in the future.
tests are also very small (see Section 5.4), so that the memory resource is not exhausted by the test program and the operating system.

The implementation of the DSM component of GENESIS does not allow all aspects of DSM to be exploited. The current state of the DSM implementation within GENESIS only allows one process to execute on one computer at the same time. This short coming means that the parallel applications generated by the tool cannot contain too many parallel processes as it is only scalable in a linear fashion. Future development within DSM will allow the tool to create a larger number of parallel processes.

5.4 Objectives and Scopes of Tests

The tests carried out on the tool were oriented towards exploring the limitations of the parallelising compiler and execution environment. The tests have been chosen to assess:

- the ability of the tool to detect an appropriate parallelism model and apply that model to the parallelisation processes
- the programmers relief from dividing a sequential program into a parallel application manually, which traditionally includes:
  - analysing the program presented
  - creating sections of the program that can be parallelised
  - setup information that is passed between parallel processes
  - synchronising the parallel processes
- the programmers relief from executing the program, which includes:
  - inserting the appropriate primitives into the parallel application
  - ensuring that the appropriate services for efficient parallel processing are available
  - mapping processes to computers
  - instantiating parallel processes on computers of a cluster
  - execution performance of the parallelised application

For each test program there are two approaches used in the assessment. Firstly the parallel application generated was examined visually. Secondly the application
was executed and the speed-up was observed.

The first approach facilitates the proof-of-usability assessment. In particular, how the programmer is assisted. The assistance includes automatic creation of parallel processes, placement of processes on computers, process instantiation and automatic execution.

The second approach is used to appraise the proof-of-performance of the tool, in particular the execution performance of the parallel application generated by the tool. To gain the elapsed time values required for this assessment the number of computers comprising the virtual machine used within the cluster was varied from one to nine\(^1\). Where applicable the size of the problem being solved (in terms of computation) was also varied, therefore allowing the performance of the parallel application to be measured.

### 5.4.1 Test programs

The two selected test programs are Towers of Hanoi and Matrix Multiplication. Each of these programs have been derived from commonly found algorithms and written in Pascal so that the parallelising compiler was able to interpret each program. The sequential programs have no extra parallel processing directives of any sort in place for the parallelising compiler to use.

These two particular test programs were selected because of the type of implementation they follow. The Towers of Hanoi application uses a recursive algorithm which when parallelised exposes the ability for the parallelising compiler to parallelise such an application. The Matrix Multiplication application contains an iterative solution with nested loops, therefore exposing the ability to exploit nested loop parallelisation.

#### 5.4.1.1 Towers of Hanoi

The Towers of Hanoi algorithm consists of a set of discs and three pegs. The

---

\(^1\) The maximum number of computers available at the time of testing.
discs decrease in size as they reach the top of the peg, hence a tower. The objective is to transfer the entire tower to one of the other pegs moving only one disk at a time and never place a larger one on top of a smaller one. This is a problem that is commonly used as an introduction to algorithms in computer science. The conventional solution to this problem is a recursive one, which is used in this test.

The algorithm chosen for the Towers of Hanoi test application in this research is presented in Figure 5.2. Extra computation is invoked in this solution by using additional looping in the main section. The extra computation is introduced to test that the parallelising compiler is able to detect a recursive section of code and loops. When recursion is detected, the compiler leaves the block of code as a parallel unit of its own. The main section of the Towers of Hanoi application contains a loop that could possibly be parallelised. The loop contains no dependencies between each iteration and therefore the 10 iterations of the loop could translate into 10 parallel processes.

```plaintext
tower(frompeg, topeg, auxpeg, num_discs) {
  if (n == 1)
    return
  else
    tower(frompeg, auxpeg, topeg, n - 1)
    echo “Moving disc “ n “ from peg “ frompeg “ to ” topeg “.”
    tower(auxpeg, topeg, frompeg, n - 1)
  endif
}

# Main Section
echo “Calling Function Now”
loop cnt from 1 to 10
  tower(‘A’, ‘B’, ‘C’, 20)
endloop
```

Figure 5.2 Pseudocode of the Towers of Hanoi program

### 5.4.1.2 Matrix Multiplication

This test program focuses on the multiplication (or cross product) of two matrices. When multiplying two matrices, the value in each row is multiplied by the value in each column. There are three matrices used in this process, the two input
matrices and one output (or result) matrix.

The common approach to parallelising this task is to partition the matrices so that each process is allocated a section to work on. The partitioning style is similar to that shown in Figure 3.3 (Chapter 3).

For the parallelising compiler to be able to partition the matrix multiplication solution automatically it must be able to analyse loops and arrays (used to implement matrices) and partition the arrays appropriately. This test program was chosen to test whether the parallelising compiler is able to partition an array so that the calculations performed on the array can be executed in parallel.

The use of shared memory also plays a part in the testing of matrix multiplication. When partitioning the arrays shared memory (and the associated services) can assist by taking care of managing the changes to the matrices. Therefore each parallel process created as part of the matrix multiplication solution is able to share a common array. The creation of the shared memory region and the divisions of the arrays can be seen with this matrix multiplication test.

The algorithm used for the Matrix Multiplication test application in this research is shown in Figure 5.3. The initialisation section can be either data read from

```plaintext
matrix_mult()
{
  initialise 1st, 2nd and results matrices
  loop row from 1 to row size of 1st matrix
    loop col from 1 to column size of 2nd matrix
      set sum to 0
      loop idx from 1 to column size of 1st matrix
        set sum to sum +
        matrix1[row][idx] *
        matrix2[idx][col]
      endloop
      set result[row][col] to sum
    endloop
  endloop
}
```

*Figure 5.3 Pseudocode for Matrix Multiplication program*
a file or statically by way of loops that initialise regular matrices. In this test case static loops are used, but a data file could quite easily be used also.

5.5 Results - Towers of Hanoi

This section presents the results observed for the Towers of Hanoi test program. Included in the observations are the qualitative and quantitative results. The testing procedures provide scope for two observations:

- that the tool is easy to use and relieves the programmer from activities of the operating system nature
- that the performance of the parallel application is acceptable or better than the sequential application.

5.5.1 Qualitative Results

The qualitative results are presented as a commentary on the parallelisation process and the parallelised application generated. The observations and expected results of the test program are also described. Implementation specific output is

```plaintext
program hanoi;
const LOOP_SIZE = 7;
var cnt : integer;

procedure tower( frompeg, topeg, auxpeg : char; n : integer );
{ Moves N discs from FROMPEG to TOPEG using AUXPEG as an auxiliary. }
begin
  if( n = 1 ) then
    break
  else
    begin
      tower( frompeg, auxpeg, topeg, n - 1 );
      tower( auxpeg, topeg, frompeg, n - 1 )
    end
  end;

begin
  for cnt := 0 to LOOP_SIZE do
    begin
      writeln(‘Calling func now!’);
      tower(‘A’, ‘B’, ‘C’, 20)
    end
  end.
```

Figure 5.4 Towers of Hanoi source program, sequential version
shown as part of these observations, including the parallel symbol table and the low level parallel application before it is compiled into machine specific code.

The sequential source program written in the Pascal language is shown in Figure 5.4. The output statements shown in the pseudo code version as echo’s (see Figure 5.2) have been removed to make the program simpler.

### 5.5.1.1 Towers of Hanoi - Message Passing based version

To begin with the parallel analysis of the parallelising compiler examines the main section and detects a loop with finite indices. The body of the loop is then marked as a possible block that can be parallelised. The loop block in this case contains a call to a routine. The parallelising compiler then examines this routine so that the possible units of parallelism can be identified and stored in the parallel symbol table, shown in Figure 5.5.

<table>
<thead>
<tr>
<th>PROC_FUNC_NAME</th>
<th>CONSTRUCT</th>
<th>CONST_NAME</th>
<th>GBL</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>MAIN</em></td>
<td>PROCEDURE</td>
<td>writeln</td>
<td>YES</td>
</tr>
<tr>
<td><em>MAIN</em></td>
<td>STANDARD_ID</td>
<td>NULL</td>
<td></td>
</tr>
<tr>
<td><em>MAIN</em></td>
<td>NULL</td>
<td>writeln</td>
<td></td>
</tr>
<tr>
<td><em>MAIN</em></td>
<td>FOR</td>
<td>NULL</td>
<td></td>
</tr>
<tr>
<td><em>MAIN</em></td>
<td>ASSIGN</td>
<td>cnt</td>
<td>YES</td>
</tr>
<tr>
<td><em>MAIN</em></td>
<td>VARIABLE</td>
<td>NULL</td>
<td></td>
</tr>
<tr>
<td><em>MAIN</em></td>
<td>CONST_ID</td>
<td>-1</td>
<td></td>
</tr>
<tr>
<td><em>MAIN</em></td>
<td>PROCEDURE</td>
<td>tower</td>
<td>YES</td>
</tr>
<tr>
<td><em>MAIN</em></td>
<td>READ (VAR)</td>
<td>tower</td>
<td>YES</td>
</tr>
<tr>
<td><em>MAIN</em></td>
<td>FOR (END)</td>
<td>NULL</td>
<td></td>
</tr>
</tbody>
</table>

tower PROCEDURE tower YES
tower READ (VAR) tower YES
tower VARIABLE NULL
tower PROCEDURE tower YES
tower READ (VAR) tower YES
tower VARIABLE NULL

**Figure 5.5** Parallel symbol table for Tower of Hanoi program

The tower routine is examined next. The first part of the if statement is not parallelisable as it is returning from the routine. The else part contains statements that could be parallelised. Each statement is examined and this reveals that there are routine calls to itself (i.e. recursive). The parallelising compiler does not delve into
Chapter 5 - Assessment of the Automated Parallel Application Creation and Execution Tool

these recursive routine calls as it is already examining the first call to the routine. As the routine calls are recursive, the tower routine cannot be automatically parallelised. Once this analysis has been completed, the only parallelisable component of this application are the calls to the tower routine within the loop of the main section.

The parallel application generated by the tool when using message passing is shown in Figure 5.6. The Tower of Hanoi program is split into three parts. The first part (Figure 5.6a) is the main or parent part of the program. This part is where the whole parallel application starts and ends. The second part (Figure 5.6b) is the first (and only in this application) parallel child which is created as part of the parallelising compilers parallel analysis. The third part (Figure 5.6c) is the routine that is called from the parallel child. This last part is separated from the second part as a generalisation to simplify the low level linking process, particularly because the low level linking is automatically generated.

```c
int main( int argc, char *argv[] )
{
    NUCprintf("%s\n", "Calling func now!");
    num_children = (__LOOP_SIZE - 0 + 1);
    child_psns = (SNAME *)malloc(sizeof(SNAME) * num_children);
    child_uports = (SNAME *)malloc(sizeof(SNAME) * num_children);
    /* Process creation primitive */
    if( proc_ncreate( "slave_1009803600_01.exe", num_children, child_psns, child_uports ) < 0 )
    {
        NUCprintf("hanoi: ERROR - process create failed\n");
        proc_terminate(0, (SNAME *)0);
    }
    num_children = (__LOOP_SIZE - 0 + 1);
    child_stats = (uint32_t *)malloc(sizeof(uint32_t) * num_children);
    returned_child_psns = (SNAME *)malloc(sizeof(SNAME) * num_children);
    proc_nwait( num_children, child_stats, returned_child_psns );
}
```

a) Parent section

```c
void main(int argc, char **argv)
{
    tower('A', 'B', 'C', 20);
    proc_terminate(0, (SNAME *)0);
}
```

b) Parallel child section

Figure 5.6 Parallel version of the Towers of Hanoi test program using message passing
The parent part of the program contains two of the primitives provided by the execution environment. The are highlighted in bold text and are `proc_ncreate()` and `proc_nwait()`. These two primitives are used to create the parallel process and synchronise the parallel application respectively.

```c
void tower( char frompeg, char topeg, char auxpeg, int n )
{
    if( (n == 1) ) {
        return;
    } else {
        tower(frompeg, auxpeg, topeg, n - 1);
        tower(auxpeg, topeg, frompeg, n - 1);
    }
}
```

c) Recursive tower routine

**Figure 5.6 (continued)** Parallel version of the Towers of Hanoi test program using message passing

### 5.5.1.2 Towers of Hanoi - Shared Memory based version

When shared memory is used, the generated parallel application is slightly different (Figure 5.7). The parent part of the application uses the `start_dsm()` primitive (Figure 5.7a) to initiate the shared memory space and create the parallel process (Figure 5.7b). The `dsm_barrier()` primitive is used to synchronise the parallel processes (both parent and child). This synchronisation is in place to allow for any changes in the shared memory space to be synchronised across all parallel processes. A similar synchronisation primitive is inserted into the child process (Figure 5.7b). The difference between the two is the parameter that defines which barrier to synchronise to. In this test program there are no changes made of the shared memory region. Therefore the synchronisation is only used to make sure that the parallel child process has attached to the shared memory region.

For the parallel child process to attach to the shared memory region, the `dsm_parstart()` primitive is used. This primitive is inserted automatically as part of the shared memory code generation, as it is required by the execution environment.

For this test program, none of the semaphore operations that the shared memory service provides were required. This is due to the parallelisation detected
within the nested loops. There was no dependencies detected between sequential sections of the program that could have been parallelised with the assistance of semaphores.

```c
int main( int argc, char *argv[] )
{
    NUCprintf("\n", "Calling func now! ");
    num_children = (__LOOP_SIZE - 0 + 1);
    dsm_1009803610_01_mem = (struct __dsm_1009803610_01_mem *)
        start_dsm("slave_1009803610_01.exe", &__dsm_sp_name,
                 RELEASE, sizeof(struct __dsm_1009803610_01_mem),
                 &num_children, __dsm_sem, __dsm_barrier, __dsm_num_sems,
                 __dsm_num_barriers);
    /* Synchronise at the start */
    dsm_barrier(__dsm_barrier[0]);
    /* Synchronise at the end */
    dsm_barrier(__dsm_barrier[1]);
    num_children = (__LOOP_SIZE - 0 + 1);
    child_stats = (uint32_t *)malloc(sizeof(uint32_t) *
        num_children);
    returned_child_psns = (SNAME *)malloc(sizeof(SNAME) *
        num_children);
    proc_nwait( num_children, child_stats, returned_child_psns );
    /* Clean up DSM barriers, locks before exiting.*/
    end_dsm();
}
```

a) Parent section

```c
void main(int argc, char **argv)
{
    dsm_1009803610_01_mem = (struct __dsm_1009803610_01_mem *)
        dsm_parstart(&__dsm_slave_num, &__dsm_sp_name,
                      &__dsm_num_procs, __dsm_num_sems, __dsm_num_barriers,
                      __dsm_sem, __dsm_barrier);
    /* Synchronise at the start */
    dsm_barrier(__dsm_barrier[0]);
    tower('A', 'B', 'C', 20);
    /* Synchronise at the end */
    dsm_barrier(__dsm_barrier[1]);
    proc_terminate(0, (SNAME *)0);
}
```

b) Parallel child section

**Figure 5.7** Parallel version of the Towers of Hanoi test program using *shared memory*

### 5.5.1.3 Qualitative Summary

The assistance to the programmer that this tool provides for the Towers of Hanoi test program can be seen in two ways:
• The decision to parallelise is made on the programmers behalf using the techniques described in the logical design and implementation of the tool. The techniques are comprised of the identification of units of parallelism, sequences and synchronisation points.
• The primitives required to perform the basic parallel tasks are inserted into the parallel application correctly and without the programmer having to learn a new language or parallel programming library.

The level of parallelism in this case was minimal but was detected and generated as expected.

### 5.5.2 Quantitative Results

The measurements taken to assess the Towers of Hanoi test program are based on the elapsed execution times of the parallel application which was executed several times. The method used to obtain the elapsed time is shown in Figure 5.8. Each time the application was executed the computation load was increased by increasing the number of discs for the three towers. The communications mechanism was also varied between message passing and shared memory.

```c
void main()
{
    /* Variable declarations */
    start_time = time_stamp();

    /* Parallel application is contained here next */
    ...

    end_time = time_stamp();
    elapsed_time = end_time - start_time;
    /* Open performance results data file and */
    /* save elapsed time */
}
```

Figure 5.8 Time stamp locations

### 5.5.2.1 Towers of Hanoi - Message Passing based version

The first sequence of tests were performed using a message passing based application. The results for this experiment are shown in Figure 5.9. A line of best fit
for all points is shown also. This line shows that although there is much variation in the observed values, the speed-up has increased as computers were added. In fact, a speed-up of nearly four (4) was observed when seven (7) computers were added to the cluster. This is an excellent outcome as the speed-up reached is quite substantial for the application size.

<table>
<thead>
<tr>
<th>Test value</th>
<th>$r^2$ Message Passing</th>
<th>$r^2$ DSM</th>
</tr>
</thead>
<tbody>
<tr>
<td>1_loop</td>
<td>0.400716</td>
<td></td>
</tr>
<tr>
<td>2_loop</td>
<td>0.799253</td>
<td>0.099696</td>
</tr>
<tr>
<td>3_loop</td>
<td>0.851735</td>
<td>0.001909</td>
</tr>
<tr>
<td>4_loop</td>
<td>0.870242</td>
<td>0.454545</td>
</tr>
<tr>
<td>5_loop</td>
<td>0.79117</td>
<td>0.983051</td>
</tr>
<tr>
<td>6_loop</td>
<td>0.811323</td>
<td>1.0</td>
</tr>
<tr>
<td>7_loop</td>
<td>0.896556</td>
<td></td>
</tr>
<tr>
<td>All tests</td>
<td>0.654713</td>
<td>0.110694</td>
</tr>
</tbody>
</table>

Table 5.1 Towers of Hanoi, coefficient of determination values for Message Passing and Shared Memory tests
The coefficient of determination for each test value is presented in the second column of Table 5.1. The values range from 40% to 89.6% and when all values are used 65.5%. Hence, we can conclude that 65.5% of the variation in the speed-up values can be explained by the linear relationship with computer numbers, whilst 34.5% of the variation in speed-up is unexplained by the model.

However, the underlying mechanisms of the execution environment do play a part in execution time and can be attributable to the unexplained variation. These mechanisms include the services and subsequently the servers of GENESIS. As more computers were added to the Virtual Machine (VM), the impact of the servers are reflected in the elapsed times.

5.5.2.2 Towers of Hanoi - Shared Memory based version

The second set of tests were performed using a shared memory based application. The results for this experiment are shown in Figure 5.10. A line of best fit for all points is also shown here. The line for this sequence of tests shows that the speed-up has slightly decreased as computers were added. The coefficient of determination for each test value is presented in the third column of Table 5.1. The values range from 0.19% to 98.3%\(^1\) and when all values are used 11.1%. Hence, we can conclude that 11.1% of the variation in the speed-up values can be explained by the linear relationship with computer numbers, whilst 88.9% of the variation in speed-up is unexplained by the model.

The reason for the variation figure of 88.9% (where the model is unable to explain the low speed-up from the idea) is hard to pin point. A preliminary investigation during testing found that the limitation of how some components were implemented as oppose to their design) were the cause of the limited speed-up. In particular, the overhead of the shared memory system (DSM) is significant [SILC98c]. This is due to how multiple individual processes that are part of a parallel application are handled on one machine of a cluster. An improvement for this could

---

\(^1\) The 100% case - 6_loop - is discounted as only two observations were made, therefore the variation will always be 0.
be an optimised re-write of the DSM system for this particular case, but is well outside the scope of this project.

The results for this shared memory test are not unexpected. The DSM system is such that it relies heavily on message passing to carry out the underlying operations needed between the various GENESIS servers [SILC97]. Therefore, as the application is quite small, the overhead of shared memory shows through. Particularly because of the high communication latency. Furthermore, the physical resources available, very small memory and a slow network both affect negatively the results because the applications must be quite small to be accommodated.

The DSM implementation on GENESIS also has the limitation that only one DSM process is allowed on the one machine at the same time. Therefore, the results for this test program using DSM start at the number of computers which reflects the number of processes created (i.e. a delayed start in the figure).

5.5.3 **Towers of Hanoi Summary**

The results observed for both message passing and shared memory with the Towers of Hanoi problem are positive. The parallelism detected and the automated
execution save the programmer a lot of work. The observed improvements of shared memory over message passing are good but the limitations of the execution environment restrict the true benefits of shared memory (reducing the number of instructions for process instantiation, data distribution and synchronisation) to shine through.

5.6 Results - Matrix Multiplication

This section presents the results observed for the Matrix Multiplication test program. Similarly to the Towers of Hanoi test program the observations include the qualitative and quantitative results.

5.6.1 Qualitative Results

The qualitative results for Matrix Multiplication are presented as a commentary on the parallelisation process and the parallelised version of the test program, similar to the Towers of Hanoi test program. The sequential source program for the Matrix Multiplication written in the Pascal language is shown in Figure 5.11.

5.6.1.1 Matrix Multiplication - Message Passing based version

In a similar fashion to the previous test program, the parallelising compiler component of the tool examines the sequential source code provided by the programmer. For this Matrix Multiplication test program, the main section is the only part to be examined. The beginning of the main section contains the initialisation of the matrices. In this test program, the initial values of the three matrices (input1, input2 and result) are set via static loops.

Other examples of the matrix multiplication process read the initialisation values form a data file. Opening files would require a Pascal library to be created, or an internal GENESIS specific primitive could be inserted within the sequential Pascal source. This would require extensions to the Pascal parser of the parallelising compiler and move away from the core aims of this research, therefore it is easier to initialise the matrices within the source program via loops, than from files.
program matrix;
const R1 = 2; R2 = 5; R3 = 2;
const C1 = 5; C2 = 2; C3 = 2;
type
  GlobalMemory = record
    input1 : array[1..R1, 1..C1] of integer;
    input2 : array[1..R2, 1..C2] of integer;
    result : array[1..R3, 1..C3] of integer;
  end;

var glob : GlobalMemory;
  row, col, i, j, sum : integer;
  first_row, last_row : integer;
begin
  for i := 1 to R1 do 
    begin
      for j := 1 to C1 do 
        begin
          glob.input1[i][j] := 1;
        end;
    end;
  for i := 1 to R2 do 
    begin
      for j := 1 to C2 do 
        begin
          glob.input2[i][j] := 2;
        end;
    end;
  for i := 1 to R3 do 
    begin
      for j := 1 to C3 do 
        begin
          glob.result[i][j] := 0;
        end;
    end;
  (* MAIN PART *)
  first_row := 1;
  last_row := R1;
  writeln('Initially, I start at ', first_row, ' and end at ', last_row);
  for row := first_row to last_row do 
    begin
      for col := 1 to C2 do 
        begin
          sum := 0;
          for i := 1 to C1 do 
            begin
              sum := sum + glob.input1[row][i] *
                  glob.input2[i][col];
            end;
          glob.result[row][col] := sum;
        end;
    end;
  writeln('Finished');
end.

Figure 5.11 Matrix Multiplication source program, sequential version
The loops used to initialise the matrices are detected, and the nested loops within are also detected. Both of these loops are marked as possible units of parallelism. This is noted in the parallel symbol table (see Figure 5.12a).

The blocks of code within each loop contain a simple assignment to initialise each element of the matrix. This is added to the parallel symbol table too. The next couple of statements in the program are assignments to define the limits on the multiplication part of the program (i.e. the next nested loop). These two statements are non intra-dependent and can therefore be parallelised too. They are added to the parallel symbol table shown in Figure 5.12b.

The next loop is the main computation loop that performs the actual multiplication. It is a nested loop also, in fact three loops deep. The loops are also

<table>
<thead>
<tr>
<th>PROC_FUNC_NAME</th>
<th>CONSTRUCT</th>
<th>CONST-NAME</th>
<th>GBL</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>MAIN</em></td>
<td>FOR</td>
<td>NULL</td>
<td></td>
</tr>
<tr>
<td><em>MAIN</em></td>
<td>ASSIGN</td>
<td>i</td>
<td>YES</td>
</tr>
<tr>
<td><em>MAIN</em></td>
<td>U_INT</td>
<td>U_REAL 1</td>
<td></td>
</tr>
<tr>
<td><em>MAIN</em></td>
<td>TO</td>
<td>NULL</td>
<td></td>
</tr>
<tr>
<td><em>MAIN</em></td>
<td>VARIABLE</td>
<td>NULL</td>
<td></td>
</tr>
<tr>
<td><em>MAIN</em></td>
<td>CONST_ID</td>
<td>R1</td>
<td></td>
</tr>
<tr>
<td><em>MAIN</em></td>
<td>FOR</td>
<td>NULL</td>
<td></td>
</tr>
<tr>
<td><em>MAIN</em></td>
<td>ASSIGN</td>
<td>j</td>
<td>YES</td>
</tr>
<tr>
<td><em>MAIN</em></td>
<td>U_INT</td>
<td>U_REAL 1</td>
<td></td>
</tr>
<tr>
<td><em>MAIN</em></td>
<td>TO</td>
<td>NULL</td>
<td></td>
</tr>
<tr>
<td><em>MAIN</em></td>
<td>VARIABLE</td>
<td>NULL</td>
<td></td>
</tr>
<tr>
<td><em>MAIN</em></td>
<td>CONST_ID</td>
<td>C1</td>
<td></td>
</tr>
<tr>
<td><em>MAIN</em></td>
<td>ASSIGN</td>
<td>input1</td>
<td>YES</td>
</tr>
<tr>
<td><em>MAIN</em></td>
<td>U_INT</td>
<td>U_REAL1</td>
<td></td>
</tr>
<tr>
<td><em>MAIN</em></td>
<td>END FOR</td>
<td>NULL</td>
<td></td>
</tr>
<tr>
<td><em>MAIN</em></td>
<td>END FOR</td>
<td>NULL</td>
<td></td>
</tr>
</tbody>
</table>

**NB: Similar entries for the input2 and results matrices**

```
... *MAIN* ASSIGN first_row YES
*MAIN* U_INT|U_REAL 1
*MAIN* ASSIGN last_row YES
*MAIN* VARIABLE NULL
*MAIN* CONST_ID R1
*MAIN* PROCEDURE writeln YES
*MAIN* STANDARD_ID NULL
*MAIN* NULL writeln
```

a) Initialisation part

**Figure 5.12 Parallel symbol table for Matrix Multiplication program**
Chapter 5 - Assessment of the Automated Parallel Application Creation and Execution Tool

added to the parallel symbol table as possibilities for parallelisation (see Figure 5.12b).

The second phase of analysis examines the initialisation loops and detects that the variables used in one set of loops are also used in the next. Therefore the two sets of loops cannot be executed together in parallel (see Figure 5.13). The inner loop blocks however can be parallelised. The nested loop is therefore flagged for parallelisation. The same inner loop parallelisation applies to the two other initialisation loops.

b) main computational part

Figure 5.12 (continued) Parallel symbol table for Matrix Multiplication program
The next two assignments are examined for inter-dependencies, there are none and therefore can be parallelised also. The next parallel unit is the main computational loop. The statements of the block of the second loop are examined. The compiler finds that the statements (an assignment, a loop and another assignment) all depend on the `sum` variable (see Figure 5.14). Therefore this loop block is kept as a whole unit, with each part executing sequentially.

The parallel application generated by the tool for the matrix multiplication test program is shown in Figure 5.15. The parent part of the program includes the initialisation loops first (see Figure 5.15a).
The nested loops (from the initialisation loops) that have been parallelised are shown in Figure 5.15b. These are the child processes instantiated by the `proc_ncreate()` primitive in Figure 5.15a, shown in bold typeface. As each initialisation loop is performing a similar task, the parallelised child applications are also very similar.

The child process immediately awaits the information from the parent process once it is created. This information consist of the values of the outer loop indices from the original sequential version. At the end of the child process, the variables modified within the child are packed into messages and sent back to the parent.

The values come from the syntax tree. If they are constants, then they can be explicitly placed in the low level code. If they are integers and symbolic, then the identifier symbols are placed in the low level code and are populated at run time. An example of the section of code that sends information to the child is shown in

```c
int main( int argc, char *argv[] )
{

    /* Initialisation of input1 matrix */
    num_children = (__R1 - 1 + 1);
    child_psns = (SNAME *)malloc(sizeof(SNAME) * num_children);
    child_uports = (SNAME *)malloc(sizeof(SNAME) * num_children);
    if( proc_ncreate("/home/chrismc/SLVS/slave_1009803620_01.exe",
                    num_children, child_psns, child_uports ) < 0 ) {
        NUCprintf("matrix: ERROR - process create failed\n");
        proc_terminate(0, (SNAME *)0);
    }

    /* Pack values to send to the parallel child process */
    /* Send values to the parallel child process */
    ... /* Receive values from the parallel child process */
    /* Extract values from the parallel child process */

    num_children = (__R1 - 1 + 1);
    child_stats = (uint32_t *)malloc(sizeof(uint32_t) * num_children);
    returned_child_psns = (SNAME *)malloc(sizeof(SNAME) * num_children);
    proc_nwait( num_children, child_stats, returned_child_psns );
    NB: This is repeated for the input2 and result matrices.
    ...
}
```

Figure 5.15 Parallel version of the Matrix Multiplication test program using message passing
When the parallel application is executing, the parent process sends information to the child processes and then waits for a response from all the children.

Figure 5.15 (continued) Parallel version of the Matrix Multiplication test program using message passing

Figure 5.16.

When the parallel application is executing, the parent process sends information to the child processes and then waits for a response from all the children.
Chapter 5 - Assessment of the Automated Parallel Application Creation and Execution Tool

After all the responses are received, the parent then waits for children to terminate. This is carried out through using the `proc_nwait()` primitive (highlighted in the second last code line of Figure 5.15a) which is used to perform the necessary synchronisation.

```c
/* Message structure "sarg" is setup here. */
__i_cnt = 1;
for( __cnt = 0; __cnt < num_children; __cnt++, __i_cnt++ ) {
    /* Set the index of the loop, this is pointed to by the assignment above. */
    i = __i_cnt;
    send( &child_uports[__cnt], &__parent_uport, sarg, &sres );
}

The address of this variable is within the sarg structure that is sent to the child process.
```

Figure 5.16 Code segment for sending information to child process

After all the responses are received, the parent then waits for children to terminate. This is carried out through using the `proc_nwait()` primitive (highlighted in the second last code line of Figure 5.15a) which is used to perform the necessary synchronisation.

```c
... first_row = 1; last_row = __R1;
NUCprintf("\"Initially, I start at \"", first_row, \" and end at \"", last_row);
num_children = (last_row - first_row + 1);
child_psns = (SNAME *)malloc(sizeof(SNAME) * num_children);
child_uports = (SNAME *)malloc(sizeof(SNAME) * num_children);
if( proc_ncreate( 
    "/home/chrismc/SLVS/slave_1009803620_04.exe", 
    num_children, child_psns, child_uports ) < 0 ) { 
    NUCprintf("matrix: ERROR - process create failed\n");  
    proc_terminate(0, (SNAME *)0); 
}  
/* Pack values to send to the parallel child process */
/* Send values to the parallel child process */
/* Receive values from the parallel child process */
/* Extract values from the parallel child process */
num_children = (last_row - first_row + 1);
child_stats = (uint32_t *)malloc(sizeof(uint32_t) * num_children);
returned_child_psns = (SNAME *)malloc(sizeof(SNAME) * num_children);
proc_nwait( num_children, child_stats, returned_child_psns );
NUCprintf("\"Finished\"\n");
```

Figure 5.17 Parallel version of the Matrix Multiplication test program using message passing
A similar process is followed for each of the matrices that are initialised. The synchronisation is required between each parallel section as the same variables are used for the loop indices.

The main computational loop is shown in Figure 5.17a. It constitutes the second major parallel section. Similarly to the parallelisation of the initialisation loops, the nested loops within this part of the program is split into a parallel child process (Figure 5.17b).

```c
void main(int argc, char **argv)
{
    /* Receive values from the parent process */
    /* Extract values from the parent process */
    ...
    for( col = 1; col <= __C2; col++ )
    {
        sum = 0;
        for( i = 1; i <= __C1; i++ )
        {
            sum = sum + glob.input1[row][i] *
                  glob.input2[i][col];
        }
        glob.result[row][col] = sum;
    }
    /* Pack values to send to the parent process */
    /* Send values to the parent process */
    ...
    proc_terminate(0, (SNAME *)0);
}
```

b) Parallel child section - main computational loop

Figure 5.17 (continued) Parallel version of the Matrix Multiplication test program using message passing

The parallel child process for the main computational loop actually contains a loop of its own along with two assignment’s. This is due to the dependency on the `sum` identifier, identified earlier (see Figure 5.14).

5.6.1.2 Matrix Multiplication - Shared Memory based version

The shared memory approach to Matrix Multiplication varies to the Tower of Hanoi test program. Due to the limitations of DSM within GENESIS, the initialisation of the `input1`, `input2` and `result` matrices cannot be parallelised into separate
parallel processes as with the message passing solution. The limitations do not allow more than one parallel section within a DSM application to occur. Therefore the initialisation of the three matrices are executed sequentially in the parent section of the program (see Figure 5.18a).

```c
int main( int argc, char *argv[] )
{
    /* Initialisation of input1, input2 and result matrices */
    for( i = 0; i <= __R1; i++ )
    {
        for( j = 0; j <= __C1; j++ )
        {
            glob.input1[i][j] = 1;
        }
    }
    for( i = 0; i <= __R2; i++ )
    {
        for( j = 0; j <= __C2; j++ )
        {
            glob.input2[i][j] = 2;
        }
    }
    for( i = 0; i <= __R3; i++ )
    {
        for( j = 0; j <= __C3; j++ )
        {
            glob.result[i][j] = 0;
        }
    }
    first_row = 1;
    last_row = __R1;
    NUCprintf("%s%d%s%d\n", "Initially, I start at ",
        first_row, " and end at ", last_row);
...```

a) Parent section

**Figure 5.18** Parallel version of the Matrix Multiplication test program using *shared memory*

The other parallel section (the main computational loop) is parallelised in a similar fashion to the message passing approach. The exception is that the shared memory region contains the whole matrix, as well as other variables that are shared between the parallel processes. The component of the generated parallel application is shown in Figure 5.18a.

The parallel child (Figure 5.18b) “receives” the information it requires for computation by attaching to the shared memory region. Both the parent and child are
then synchronised via the inserted dsm_barrier() primitive - shown in Figure 5.12a (continued) and Figure 5.12b as the first bold dsm_barrier() primitive. Any reference to change part of the shared memory space is placed within the lock() & unlock() primitives. These two primitives use a semaphore to signify when the execution is allowed to continue. The semaphores are created as part of the low level code the tool generates. The DSM services of GENESIS carry out the

```
...
num_children = (last_row - first_row + 1);
dsm_1009803630_01_mem = (struct __dsm_1009803630_01_mem *)
start_dsm("/home/chrismc/SLVS/slave_1009803630_01.exe",
          &__dsm_sp_name, RELEASE, sizeof(struct
          __dsm_1009803630_01_mem), &num_children, __dsm_sem,
          __dsm_barrier, __dsm_num_sems, __dsm_num_barriers);
/* Populate the dsm_1009803630_01_mem pointer here, so that
** the slave process(es) get the correct starting values. */
dsm_1009803630_01_mem->col = col;
dsm_1009803630_01_mem->sum = sum;
dsm_1009803630_01_mem->i = i;
memcpy(&(dsm_1009803630_01_mem->glob), &glob, sizeof(struct
globalmemory));
dsm_1009803630_01_mem->row = row;
/* dsm structure all complete now */

/* Synchronise at the start */
dsm_barrier(__dsm_barrier[0]);
/* Synchronise at the end */
dsm_barrier(__dsm_barrier[1]);

/* Extract DSM values from dsm_1009803630_01_mem here */
col = dsm_1009803630_01_mem->col;
sum = dsm_1009803630_01_mem->sum;
i = dsm_1009803630_01_mem->i;
memcpy(&glob, &(dsm_1009803630_01_mem->glob), sizeof(struct
globalmemory));
row = dsm_1009803630_01_mem->row;
/* DSM structure deconstructed now */
num_children = (last_row - first_row + 1);
child_stats = (uint32_t *)malloc(sizeof(uint32_t) *
            num_children);
returned_child_psns = (SNAME *)malloc(sizeof(SNAME) *
            num_children);
proc_nwait( num_children, child_stats, returned_child_psns );
NUCprintf("%sn", "'Finished'"Mitch);  /* Clean up DSM barriers, locks, and other stuff before
** exiting.*/
end_dsm();
}
```

a) (continued) Parent section

**Figure 5.18 (continued)** Parallel version of the Matrix Multiplication test program using shared memory
In terms of the qualitative assessment, the tool has assisted the programmer by detecting the parallelism available within the Matrix Multiplication test program. The matrices used for the multiplication calculations are able to be parallelised by dividing up the matrix so that several child parallel processes perform the appropriate calculations required.

As for the Tower of Hanoi test program, the programmer is assisted greatly by the automated insertion of primitives into the parallelised program, that allow the characteristics of the execution environment to be exploited to its fullest potential.

Figure 5.18 (continued) Parallel version of the Matrix Multiplication test program using shared memory
5.6.2 Quantitative Results

As for the Tower of Hanoi test program, the elapsed time of the parallel application is the unit of measure used in the performance based tests. This testing method was repeated several times, each time with a different matrix size and number of computers within the virtual machine.

5.6.2.1 Matrix Multiplication - Message Passing based version

For the first set of tests Message Passing based application was used. There were two versions of the tests carried out. The first version had only one parallel section of code. This is so that a comparison can be made between Message Passing and Shared Memory, which has the stipulation that only one parallel section can be created for the whole parallel application (explained in Section 5.6.1).

The results of this test are shown in Figure 5.19. A line of best fit for all points is shown also. This line shows that the speed-up values are increasing even though there is quite some variance in the values. The line of best fit does not include the one computer case in its calculations.

![Figure 5.19 Matrix Multiplication - Message Passing (single parallel section)](image)

The coefficient of determination for each test value is presented in the second
column of Table 5.2. The values range from 1.5% to 67% and when all values are used 3.67%. Hence, we can conclude that 3.67% of the variation in the speed-up values can be explained by the linear relationship with computer numbers, whilst 96.3% of the variation in speed-up is unexplained by the model.

A particularly interesting observation is the convergence of all values except the 2x20 case. The 2x20 case was unusual in that it provided a speed-up for all computer numbers, starting at one. This occurred because the computation specified by the parallel application passing a threshold. The overheads of the GENESIS services define this threshold. When the computation time for a process is greater than the time for a service to be performed, the 2x20 situation occurs.

The second version of the test using Message Passing had multiple sections of parallel code. This version allows the full parallelisation of the tool to be analysed. The results for this test are shown in Figure 5.20. A line of best fit is included within this graph. This line shows a similar increase in speed-up to the single parallel process section case. Once again the one computer case is not included in the best fit calculations.

<table>
<thead>
<tr>
<th>Test value</th>
<th>Message Passing - single parallel section</th>
<th>Message Passing - multiple parallel section</th>
<th>DSM</th>
</tr>
</thead>
<tbody>
<tr>
<td>2x5</td>
<td>0.042634</td>
<td>0.426902</td>
<td>0.104934</td>
</tr>
<tr>
<td>2x10</td>
<td>0.106171</td>
<td>0.104147</td>
<td>0.33391</td>
</tr>
<tr>
<td>2x20</td>
<td>0.676283</td>
<td>0.163635</td>
<td>0.0922</td>
</tr>
<tr>
<td>5x5</td>
<td>0.020685</td>
<td>0.445018</td>
<td>0.534549</td>
</tr>
<tr>
<td>5x10</td>
<td>0.080937</td>
<td>0.007176</td>
<td>0.984728</td>
</tr>
<tr>
<td>5x20</td>
<td>0.015012</td>
<td>0.22932</td>
<td>0.553622</td>
</tr>
<tr>
<td>All tests</td>
<td>0.036654</td>
<td>0.019804</td>
<td>0.029244</td>
</tr>
</tbody>
</table>

*Table 5.2 Matrix Multiplication, coefficient of determination values for Message Passing and Shared Memory tests*
The coefficient of determination for each test value of this test is presented in the third column of Table 5.2. The values range from 0.7% to 44.5% and when all values are used 2.0%. Hence, we can conclude that 2.0% of the variation in the speed-up values can be explained by the linear relationship with computer numbers, whilst 98% of the variation in speed-up is unexplained by the model.

The speed-up of the message passing tests for Matrix Multiplication was positive but not as steep as the Towers of Hanoi message passing test. This is due to the amount of communication required for the Matrix Multiplication application when parallelised. The communication overhead of the execution environment of GENESIS is therefore a factor of the speed-up results.

The scalability of the testing was also limited by the size of the cluster available. The small number of computers available for the cluster and the physical memory size of each computer affects the application size and therefore the speed-up results. Physical memory also effected the size of the matrices used in the testing. The communication, cluster size and memory size factors emphasise the overheads of the execution environment.
For both versions of the test, once two computers exist in the cluster, there is a noticeable decrease in the speed-up. This decrease is due to the influence of remote communications against the size of the parallel application. After this point however, the parallelisation within the application begins to show positive improvement. The increase is attributable to the parallelism detected and inserted in the generated parallel application by the tool. This is the reason that the line of best fit does not include the one computer case. If the one computer case was included, the line of best fit would always be negative.

### 5.6.2.2 Matrix Multiplication - Shared Memory based version

The second set of tests were performed on the application developed using shared memory. The results are shown in Figure 5.21. A line of best fit for all points is also shown here. The line for this sequence of tests shows that the speed-up has decreased slightly as computers were added. The coefficient of determination for each test value is presented in the fourth column of Table 5.2. The values range from 9.2% to 98.5% and when all values are used 2.9%. Hence, we can conclude that 2.9% of the variation in the speed-up values can be explained by the linear relationship with computer numbers, whilst 97.1% of the variation in speed-up is unexplained by the model.

![Figure 5.21 Matrix Multiplication - Shared Memory](image)
As with the Towers of Hanoi test program, the decrease of speed-up for the shared memory tests is related to the slow network and the inherent overhead of using DSM. Also, the amount of physical memory available also contributed to the emphasis shown when using the shared memory paradigm. The communication requirements of the Matrix Multiplication application are also brought out because of the slow network that is used.

The limitation of the DSM implementation shown with the Towers of Hanoi test program must also be adhered to with the Matrix Multiplication test program. The two starting points (as seen in Figure 5.21) for tests with DSM are at two and five computers. This is because only one DSM process can execute on one computer for the execution life of the parallel application. So when five processes are created with the $5 \times 5$, $5 \times 10$ and $5 \times 20$ matrices, the test program cannot be executed on a virtual machine of only two, three or four computers. Despite this, three different computational loads were incurred for the matrices with 2 and 5 rows. For the 2 rows case, the speed-ups increased for each. For the 5 rows case the number of processes and inherent communications induce a reduction in the performance of the program.

### 5.6.3 Matrix Multiplication Summary

The performance for the Matrix Multiplication test program is similar to the Towers of Hanoi test program, when comparing the Message Passing and DSM results for each. Both Message Passing and DSM paradigms presented favourable results for qualitative and quantitative analysis. The programmer is relieved from the parallelisation which includes splitting up the matrices used for multiplication for both paradigms. When the Message Passing paradigm is used, the specific information required to be passed between parallel processes is carried out automatically by the parallelising compiler, therefore the programmer is relieved of this task. The overhead of the DSM paradigm and the implementation issued within the GENESIS execution environment influenced the Matrix Multiplication test program. The effect of this was shown in the quantitative results. However, the programmer is relieved from the specific setup and use of the shared memory space throughout the parallel application.
5.7 Summary

The aims of this chapter were to provide the proof-of-usability and proof-of-performance of the tool. This demonstration was carried out through qualitative and quantitative methods. The objectives of this assessment was to find out how much the tool assisted the programmer towards complete automation of parallelisation and execution. The testing also allows the limitations of the parallelising compiler and execution environment to be shown.

The tests were conducted using the GENESIS execution environment, with both Message Passing and Distributed Shared Memory being used. There were several hardware and software limitations identified as a result of testing. The hardware limitations include: the scalability of the physical hardware, memory limitations, and high communication latency. The software limitation consists of the restrictive DSM implementation within GENESIS (as stated in Section 5.3.2), the effects of these limitations were revealed in the test results. Future enhancements within the DSM implementation will remove the software limitations identified in this research. The future development of an i386\textsuperscript{1} version of GENESIS will remove most (if not all) of the hardware limitations identified.

The GENESIS implementation of DSM has provided poor performance observations for our tests. This is particularly pertinent when two DSM parallel processes are executing on the one physical computer (i.e. sharing the same PCU and other local resources). An optimisation of the DSM so that the memory updates (or “diffs” [SILC98b]) are sent via physically shared memory would improve performance greatly. This is due to direct memory management being faster than via other several layers of processes and/or external communications. If this optimisation was in place, several parallel processes using DSM executing on the one physical computer should show a reduced performance loss that is currently observed.

Both qualitative and quantitative tests were applied to two test programs (Tower of Hanoi and Matrix Multiplication). In terms of the qualitative assessment, the tests showed that the programmer was assisted greatly. Analysis, parallelisation,
communications and execution were all performed automatically for the programmer. No extra input was required from the programmer at all. In relation to the quantitative assessment, the test results varied.

The qualitative analysis showed that for the Towers of Hanoi program, the tool is able to detect recursion but not parallelise it. The parallel processes are derived from the main loop that calls the recursive function. In contrast, the Matrix Multiplication program showed that nested loops are able to be detected and parallelised. In both cases the programmer was assisted by carrying out parallelisation and execution automatically.

The quantitative analysis demonstrates that for both test programs there are improvements produced by the parallelisation that the tool detects and generates. The very small size of the test programs and the limited resources of the execution environment are important factors to consider when examining the quantitative results. For Towers of Hanoi, when using the Message Passing mechanism, the speed-up increased, while when the Shared Memory mechanism was used, there was a slight decrease in speed-up. For Matrix Multiplication, the results were similar, with the Message Passing paradigm achieving an increase in speed-up, and the Shared Memory paradigm showing a slight decrease in speed-up. The general trend of both test programs has shown positive results, vindicating the parallelisation and execution produced by the tool.

From the programmers point of view, the results of the two test programs are positive. If a programmer were to parallelise the programs manually, using the DSM paradigm would prove to be easier as the DSM paradigm is simpler by nature to use. Adopting the Message Passing paradigm would require the programmer to learn even more. The manual approach is very time consuming, error prone and expensive. In other research into automated or assisted parallelisation, the results have not been fully satisfactory [PADU96] from the programmers point of view. Whereas our research has produced results that are very satisfactory, primarily because the whole process is automated, not requiring the programmer to provide extra information at all.
There is no comparison made with other manual or semi-automatic systems due to the lack of published performance output or benchmarking of practical systems in this area of research and no common hardware and software to make comparisons. Therefore the only comparison that could be made was via our own analysis of the performance speed-up.

The assessment results presented in this chapter demonstrate the ability of the tool to automatically parallelise and execute a sequential program. The tool is very usable as the programmer has only to provide a sequential program to achieve the benefits of parallel processing. Execution of the generated parallel application is carried out by the tool also which adds to the usability of the tool from the programmers point of view. The performance to the tool is acceptable and is only limited by the execution environment hardware and software.
Chapter 6 Conclusions and Future Work

In this chapter, the contributions made to the automatic development and execution of parallel applications by this research are summarised. Our conclusions are also presented as well as possible future research.

6.1 Research Outcomes and Conclusions

Creating a parallel application is currently not an easy task. There are many complexities involved that require the knowledge of skilled programmers. This implies that the development of parallel applications has been carried out by a very small group of specialists, excellent research programmers. The cost of this process is very high and development time is long. Thus, only scientific and advanced engineering problems are solved using parallel systems. A push to parallelise problems is strong and the benefits are often large.

Cost of execution of parallel applications has been considerably high in the past due to the massively parallel processor (MPP) environments required to execute parallel applications that justify the high cost. Alternate more adaptive technologies need to be used. The area of research examining these technologies is cluster based parallel processing. Execution upon a cluster requires similar programming skills to a dedicated massively parallel execution environment, but a well designed cluster execution environment should be efficient and easy to use. Issues, such as process mapping and resource sharing need to be resolved by the programmer, but with an automated solution, these issues are solved.

Our research has identified several different approaches to parallel application design and parallel application execution. These approaches range from completely manual (based around libraries) to semi-automatic (tools). The manual methods require the programmer to be involved in design and execution of a parallel application. For the semi-automatic methods, the parallel application design and creation is made easier through a library that facilitates process management as well
as setting up an execution environment, albeit manually.

Regardless of the method used to create a parallel application, the approaches found for the execution of a parallel application be it MPP or cluster based, also rely on the programmer. Specifically, the initialisation of the execution environment so that it is ready for parallel processing and the instantiation of the parallel application upon the execution environment. This implies that there is a high threshold for parallel processing to become a part of mainstream computing. This threshold is more serious in today’s environment where there are many tasks in business, management and engineering which could greatly benefit from parallel processing. Therefore, a solution that reduces the cost of development and execution of parallel applications so that the benefits of parallel processing are more easily obtainable is required.

As a result of these findings, the focus for this research is concerned with the total automation of the development of the parallel application and its execution. This automation will support ordinary users such as managers, accountants, engineers and researchers.

The aim of this research was build a technology to simplify the development and execution of parallel applications. The ordinary programmer should not have to provide any extra input in either development or execution of a parallel application. This aim has been achieved through the completion of the following tasks:

1) Construction of a research parallelising compiler capable of automatic parallelisation and generation of information for a parallel execution environment, including:
   – several phases of a parallelising compiler working together coherently
   – each phase identifying a different component for the required parallel analysis
   – generating internal data structures to pass internally during analysis, forming information to be used to create the parallel application
   – producing information for automatic instantiation and execution of a parallel application
2) Identifying and utilising an execution environment that provides:
   - transparent operation and provision of services needed for parallel processing
   - maximum exploitation of system resources via a cluster based distributed execution environment
   - suitability for coarse grain parallelism
3) Linking of the parallel application to the execution environment, which:
   - relieves the programmer from the execution environment related tasks such as environment setup, process mapping, parallel process instantiation and parallel application management
   - provides completely automated execution

To achieve the research aims, methodology, methods and techniques of experimental computer science were applied. Three experimental computer science methodologies have been used in reaching the aims of this research: proof-of-concept, proof-of-usability and proof-of-performance. The design and implementation of the automated parallel creation tool followed proof-of-concept techniques. The concept was the design of the parallelising compiler and its integration with the execution environment. This concept was proved via the implementation of this design. To test the implementation both the usability and performance were addressed. The usability analysis examined what benefits the programmer received in terms of the creation and execution of a parallel application. The performance analysis examined the speed-up of two test applications and presented observations of those speed-ups. The usability and performance analysis constitute the qualitative and quantitative measures of the tool.

The parallelising compiler was designed in a modular fashion with six phases. Each phase performs some analysis and generates information which is used in the next phases. The information generated is contained within a symbol table, syntax tree and a parallel symbol table. The symbol table and syntax tree are created in Phase 1 and are used throughout all phases. The parallel symbol table is generated in Phase 2 and used in Phase 3. The syntax tree forms the backbone of information used to generate the parallel application. The parallel symbol table is an additional data structure that is used to assist the detection of parallel sequences. This structure
simplifies the detection process. Phases 3 and 4 are responsible for identifying sequences and synchronisation points within the application. Phase 5 performs some low level optimisation and Phase 6 generates the low level code required before execution of the parallel application. Each phase is responsible for the tasks prescribed to it and nothing else (i.e. the modular aspect of the design). Additional phases can easily be added in the future if required because of this modular phase based approach. Therefore, the parallelising compiler consists of traditional compiler related tasks (lexical, syntax and semantic analysis, optimisation and low level code generation), and parallel analysis.

The parallelising compiler itself has been implemented in a procedural environment. Each phase exists as a procedure which revolves around the syntax tree data structure that is traversed within each phase. The information each phase uses comes from the elements of the syntax tree data structure. Various elements are modified as each phase conducts its task. These elements then allow each subsequent phase to carry out their analysis.

The information produced by the parallelising compiler is utilised by the execution environment for automatic parallelisation. The requirements of the parallelising compiler and the execution environment are both satisfied as part of the parallel application/execution environment linking process. The GENESIS execution environment provides access to the required services via a suite of primitives. It is these primitives that are inserted into and contained within the parallel application generated by the parallelising compiler.

To insert the correct primitives within the parallel application, the information generated and stored within the syntax tree is used. The parameters needed to utilise the specific primitives are gathered from the information contained within the syntax tree also. The primitives are specifically used to: create processes, provide shared memory regions, provide message passing mechanisms, synchronisation and semaphore services.

To achieve total automation of parallel execution the execution environment must provide services to support parallel processing tasks that the programmer would
normally be involved with. These services include: handling the allocation of names to processes for communication and process management, mapping processes to computers and balancing the computational and communication loads of the execution environment. When left with the programmer, these tasks are both mundane and complex. The selection of an execution environment that provides the required services automatically is critical for this research. For this reason GENESIS has been selected.

The servers within GENESIS act upon service requests invoked via the primitives automatically inserted into a parallelised application. All the coordination of resources are handled by these servers. For example, one or more parallel processes are created through one simple primitive. From that point on the execution environment handles all coordination of the process(es). The selected execution environment therefore relieves the programmer from the cumbersome and error prone initialisation and management issues that are present in existing systems.

The implementation of the proposed parallelising compiler and the integration of the generated parallel application to the execution environment provide a proof-of-concept. This implementation demonstrates clearly that the design of the parallelising compiler provides parallel applications that are suitable for the desired execution environment. In addition, linking the parallel application to the execution environment provides a completely automatic solution. This is because all the parallel processing instructions are self contained within the parallel application. The execution environment provides the servers that act on the instructions from the parallel application without human interaction. Therefore, the programmer no longer has to be concerned about creating parallel applications and their execution.

Proof-of-usability is shown by the assistance the tool provides the programmer. This aspect of testing showed that parallelism was detected, a parallel application was automatically generated, the execution environment was linked and the parallel application was executed. The programmer was only required to specify which paradigm to use throughout the parallel analysis process. The paradigms that were available for selection are message passing or shared memory. This was specified at the command line of the tool and along with the original sequential
Chapter 6 - Conclusions and Future Work

application is the only input required from the programmer. All stages of the parallelisation process were completed automatically without any extra programmer input.

Proof-of-performance is shown by observing the execution of the tool as quantifiable results of elapsed execution times. The results of two test programs as a part of a performance study, showed that the speed-up of the parallel applications created are good in the case of message passing based applications. They also clearly demonstrated that DSM based applications require better clusters consisting of more powerful PCs and faster networks and will benefit from larger application sizes. The performance study also shows that by adopting message passing or DSM does effect the performance, as expected. The message passing based solutions require more information to be specified by the tool, in particular primitives to send and receive critical data to parallel processes. However the parallelising compiler generates all that is necessary automatically. When using DSM, the opportunity emerges for reducing the tasks the parallelising compiler must perform explicitly especially when communicating values of identifiers and data structures. Semaphore operations are also available when using shared memory and enable further parallelism to be detected and exploited. This further emphasises the value of the services provided by the GENESIS execution environment.

In conclusion, this research has produced a technology, which provides automated parallelisation and execution that other systems have not previously addressed. A proof-of-concept in the form of a tool that parallelises and executes applications automatically shows that the technology is feasible. The novel approach of linking the parallelising compiler to the GENESIS execution environment has proven to be quite successful. In particular, the services of GENESIS, accessed through uncomplicated primitives, permit easier coupling of the parallelising compiler to the execution environment. Through linking these two entities together, our tool dismisses the requirement for the programmer to write parallel applications and provide an environment for their execution. Further details of this work has been published in [MCAV00] and [MCAV01].

The development, implementation, execution and testing of this tool have
allowed the aims of the research to be achieved. By using our tool the programmer does not have to worry about performing parallel analysis or any execution environment related tasks. Without such a tool, programmers must continue to be trained with specific parallel processing skills.

### 6.2 Future work

The modular design of the automated parallel creation and execution tool has simplified the implementation of the tool. The tool generates parallel applications which are executed on a cluster based execution environment and are designed to exploit the resources of the execution environment as much as possible. As with any project, the scope can always be increased to accommodate more ideas and modifications, this project is no exception.

One idea would be the use of other non procedural languages for the source “sequential” program. Another idea can be derived from the proof-of-performance studies. Those studies highlighted some deficiencies in the execution environment implementation, in particular the distributed shared memory system. Future research to address the limitations and weaknesses of the tool are presented in this section.

The existing parallelisation capabilities of the parallelising compiler component of the whole tool form a skeleton. Many of the parallelisation techniques that have been developed around the world could easily be introduced. The modular design of the phases within the compiler permits this. Further research into the performance of this tool would need to take advantage of this.

The current method of disseminating information to child parallel processes is selected by the programmer at the command line when the tool is to be executed. This command line instruction specifies either the Message Passing or Distributed Shared Memory paradigm which is then used throughout the application. This does not allow the most optimal method to be applied, because different applications are more suited to a certain paradigm. Therefore, further development into the automatic selection of one paradigm or the other is required. To automate the selection of a paradigm the necessary analysis would need to be included to determine which paradigm is more
suitable. The decision could then be stored within the syntax tree to notify the low level code generation phase to generate the code appropriately. Therefore, including such automation would remove the need for the programmer to select an appropriate paradigm and also provide more parallelisation of an application than is currently available when selecting a paradigm manually.

Non procedural languages such as Object Oriented (OO) languages, would be another area where further research into the area of automated parallelisation could be carried out. A specific language such as Java or C# would provide scope for this via the class/object structure of applications written in these languages. In fact the OO structure could be very closely aligned to coarse grain parallelism. Exploration of this type of language structure for parallelisation would be an interesting project for future work.

The limitations of the execution environment (primarily the ratio of processing speed to communication latency) are emphasised by the relatively small amount of computation that the generated test parallel applications provide. This is particularly emphasised when loops are parallelised using the existing parallel analysis implemented. To improve this, larger loop blocks could be used to take advantage of the coarse granularity of the cluster based execution environment. The loop block sizes could also be made proportional to the size of the resources available. By adopting such a scheme the number of parallel process would also be decreased. Under such a scheme the parallel processes themselves would carry out more of the loop infrastructure as well.

Future work into the GENESIS DSM system to explore optimisations into more efficient use of physically shared memory on one computer should be carried out. Multiple parallel processes using DSM would then be able to execute on the same computer without the performance deterioration as currently observed. The optimisation should look at the overheads that are caused by the differences in memory spaces that are shuffled around. When multiple parallel processes are attached to the same memory space via DSM, direct memory management should be used. Higher level processing could be removed in this case. This enhancement would provide scope for seamless migration of processes around the cluster so that the
absolute maximum exploitation of system resources is achieved.
Bibliography


Bibliography


[ROZI92] Rozier, M., Abrossimov, V., Armand, F., Boule, I., Gien, M., Guillemont, M., Herrmann, F. e. e., Kaiser, C., Langlois, S., L'eonard,


[SILC99] Silcock, J. and Goscinski, A., *A comprehensive distributed shared memory system that is easy to use and program*, Distributed Systems and Engineering, 6, 1999, 121-128.


International Conference on Supercomputing, Vienna, Austria, 1997, 156-163.


Appendix A
Example Makefile for the Matrix Multiplication Parallel Application

HOST=$(shell uname -n)
ifneq ($(HOST), wallace)
    include /home/mick/3MTRhodos/config/Makefiles/sun3.defs
endif
CC = sun3-gcc
NM = sun3-nm

DEPEND_CPPFLAGS := $(CPPFLAGS) -I/home/mick/3MTRhodos/user/chrismc/include
CPPFLAGS := $(CPPFLAGS) -I/home/mick/3MTRhodos/user/chrismc/include -Wall
SPECIALIB := /home/mick/3MTRhodos/user/chrismc/lib/libchrismc.a
LDFLAGS := $(LDFLAGS) -L/home/mick/3MTRhodos/user/chrismc/lib/
LIBS := $(LIBS) -lchrismc

matrixOBJS = matrix.o matrix_var.o
slave_1043051400_01OBJS = slave_1043051400_01.o matrix_var.o
slave_1043051400_02OBJS = slave_1043051400_02.o matrix_var.o
slave_1043051400_03OBJS = slave_1043051400_03.o matrix_var.o
slave_1043051400_04OBJS = slave_1043051400_04.o matrix_var.o

PROGRAM = matrix
SRCS = $(matrixOBJS:.o=.c)
all: $(PROGRAM) \
    /home/mick/3MTRhodos/unix/root/home/chrismc/SLVS/slave_1043051400_01.exe \
    /home/mick/3MTRhodos/unix/root/home/chrismc/SLVS/slave_1043051400_02.exe \
    /home/mick/3MTRhodos/unix/root/home/chrismc/SLVS/slave_1043051400_03.exe \
    /home/mick/3MTRhodos/unix/root/home/chrismc/SLVS/slave_1043051400_04.exe

$(PROGRAM): Makefile.slave $(matrixOBJS) $(SPECIALIB)
    $(CC) $(LDFLAGS) $(CPPFLAGS) $(UPROC_HEADER) \
    $(matrixOBJS) \n    -o $@ \n    $(LIBDIRS) $(LIBS)
$(NM) -u $@
$(CC) -s -d -nostartfiles -nostdlib $(CFLAGS) \n    -Xlinker -Ttext -Xlinker 2020 \n    -Xlinker -Tdata -Xlinker 20000 \n    -o $@.exe matrix
$(NM) -u $@.exe 

/home/mick/3MTRhodos/unix/root/home/chrismc/SLVS/
slave_1043051400_01.exe: $(slave_1043051400_01OJBJS)
 $(CC) $(LDFLAGS) $(CPPFLAGS) $(UPROC_HEADER) \ 
 $\_slave_1043051400_01OJBJS) \ 
 -o slave_1043051400_01 \ 
 $(LIBDIRS) $(LIBS)
 $(NM) -u slave_1043051400_01
 $(CC) -s -d -nostartfiles $(CFLAGS) \ 
 -Xlinker -Ttext -Xlinker 2020 \ 
 -Xlinker -Tdata -Xlinker 20000 \ 
 -o $@ slave_1043051400_01
 $(NM) -u $@

/home/mick/3MTRhodos/unix/root/home/chrismc/SLVS/
slave_1043051400_02.exe: $(slave_1043051400_02OJBJS)
 $(CC) $(LDFLAGS) $(CPPFLAGS) $(UPROC_HEADER) \ 
 $\_slave_1043051400_02OJBJS) \ 
 -o slave_1043051400_02 \ 
 $(LIBDIRS) $(LIBS)
 $(NM) -u slave_1043051400_02
 $(CC) -s -d -nostartfiles $(CFLAGS) \ 
 -Xlinker -Ttext -Xlinker 2020 \ 
 -Xlinker -Tdata -Xlinker 20000 \ 
 -o $@ slave_1043051400_02
 $(NM) -u $@

/home/mick/3MTRhodos/unix/root/home/chrismc/SLVS/
slave_1043051400_03.exe: $(slave_1043051400_03OJBJS)
 $(CC) $(LDFLAGS) $(CPPFLAGS) $(UPROC_HEADER) \ 
 $\_slave_1043051400_03OJBJS) \ 
 -o slave_1043051400_03 \ 
 $(LIBDIRS) $(LIBS)
 $(NM) -u slave_1043051400_03
 $(CC) -s -d -nostartfiles $(CFLAGS) \ 
 -Xlinker -Ttext -Xlinker 2020 \ 
 -Xlinker -Tdata -Xlinker 20000 \ 
 -o $@ slave_1043051400_03
 $(NM) -u $@

/home/mick/3MTRhodos/unix/root/home/chrismc/SLVS/
slave_1043051400_04.exe: $(slave_1043051400_04OJBJS)
 $(CC) $(LDFLAGS) $(CPPFLAGS) $(UPROC_HEADER) \ 
 $\_slave_1043051400_04OJBJS) \ 
 -o slave_1043051400_04 \ 
 $(LIBDIRS) $(LIBS)
 $(NM) -u slave_1043051400_04
 $(CC) -s -d -nostartfiles $(CFLAGS) \ 
 -Xlinker -Ttext -Xlinker 2020 \ 
 -Xlinker -Tdata -Xlinker 20000 \ 
 -o $@ slave_1043051400_04
 $(NM) -u $@

clean:
 /bin/rm -f *.o *.reloc symbol_table liel_symtab.tree tcl_tree
 $(PROGRAM) slave_1043051400_01 slave_1043051400_02
 slave_1043051400_03 slave_1043051400_04
clean_all: clean
    /bin/rm -f *.exe *.c *.h

depend:
    makedepend $(DEPEND_CPPFLAGS) $(SRCS)
Appendix B  Example Execution
Batch File

#!/bin/bash
# Execution batch script for the "matrix" parallel application
cd ~/parallel_applications/matrix
make -f Makefile.matrix
if [ $? -eq 0 ]; then
   scc¹ -auto_kbuild rhodos-3a -send_ctrlc -def matrix_defaults_file
fi

1. Serial Connection Client - a GUI to interact with all computers of the GENESIS cluster.
Appendix C  Message Passing Parent Example

matrix.c

/* Include Files */
#include <sname.h>
#include <portname.h>
#include <ipc.h>
#include <rhodos.h>
#include <time.h>
#include <rufs/iface.h>
#include <rufs/file_cntl.h>
#include "matrix.h"

timeval sub_time(timeval *, timeval *);

int main( int argc, char *argv[] )
{
    int32_t num_children;
    uint32_t *child_stats;
    SNAME *returned_child_psns;
    SNAME *child_psns, *child_uports, child_info;
    int32_t child_return_value, child_cnt;

    /* Needed for testing */
    int32_t num_hosts;

    timeval start_time, stop_time, diff_time;

    /* Execution environment bug fix */
    int *z_z;
    z_z = (int *)malloc(sizeof(z_z));

    /* Performance skeleton */
    if( argc == 2 )
    {
        num_hosts = atoi(argv[1]);
    }
    else
    {
        num_hosts = 999999;
    }

    get_uport( &__parent_uport );

    /* Start time stamp */
    get_up_time( &start_time );

    num_children = (__R1 - 1 + 1);
    child_psns = (SNAME *)malloc(sizeof(SNAME) * num_children);
    child_uports = (SNAME *)malloc(sizeof(SNAME) * num_children);
    if( proc_ncreate( "/home/chrismc/SLVS/slave_1043051400_01.exe",
                     num_children, child_psns, child_uports ) < 0 )
    {
        NUCprintf("matrix: ERROR - process create failed\n");
        proc_terminate(0, (SNAME *)0);
    }
}
/ * Argument sending bit */
{
    SE_ARG *sarg, *ptr;
    SRESULTS sres;
    int __cnt, __i_cnt;
    sarg = ptr = (SE_ARG *)malloc(sizeof(SE_ARG));
    ptr->sa_options = OP_MESSAGE;
    ptr->sa_next = (SE_ARG *)0x0;
    ptr->sa_buffer = (char *)&j;
    ptr->sa_size = sizeof(int);
    if(!ptr->sa_next)
        {ptr->sa_next = (SE_ARG *)malloc(sizeof(SE_ARG));
         ptr = ptr->sa_next;
        }
    ptr->sa_buffer = (char *)&(glob.input1);
    ptr->sa_size = sizeof(glob.input1);
    if(!ptr->sa_next)
        {ptr->sa_next = (SE_ARG *)malloc(sizeof(SE_ARG));
         ptr = ptr->sa_next;
        }
    ptr->sa_buffer = (char *)&i;
    ptr->sa_size = sizeof(int);
    if(sarg->sa_next)
        {sarg->sa_options |= OP_MULTIBUF;
        }
    __i_cnt = 1;
    for(__cnt = 0; __cnt < num_children; __cnt++, __i_cnt++)
        { /* Set the index of the loop, this is pointed to by the
           ** assignment above. */
            i = __i_cnt;
            send(&child_uports[__cnt], &__parent_uport, sarg,
                 &sres);
        }
}
/* Argument receiving bit */
{
    RE_ARG *rarg, *ptr;
    RRESULTS rres;
    SNAME __rtn_port;
    int __cnt, __array_idx, __uport_cnt;
    /* malloc() off memory to receive into. This portion of memory
    ** needs to be large enough to hold the complete values sent
    ** back, which is then dissected into the actual components that
    ** the particular child calculated/changed. */
    ptr = rarg = (RE_ARG *)malloc(sizeof(RE_ARG));
    ptr->ra_options = OP_MESSAGE;
    ptr->ra_next = (RE_ARG *)0x0;
    ptr->ra_buffer = (char *)malloc(sizeof(int));
    ptr->ra_size = sizeof(int);
    if(!ptr->ra_next)
        {ptr->ra_next = (RE_ARG *)malloc(sizeof(RE_ARG));
         ptr = ptr->ra_next;
        }
    /* NEED TO MAKE SURE THAT THIS IS THE CORRECT SIZE! */
    ptr->ra_buffer = (char *)malloc(sizeof(glob.input1));
    ptr->ra_size = sizeof(glob.input1);
    if(!ptr->ra_next)
        {ptr->ra_next = (RE_ARG *)malloc(sizeof(RE_ARG));
         ptr = ptr->ra_next;
        }
}
ptr->ra_buffer = (char *)malloc(sizeof(int));
ptr->ra_size = sizeof(int);
if( rarg->ra_next ) {
    rarg->ra_options |= OP_MULTIBUF;
}

for( __cnt = 0; __cnt < num_children; __cnt++ ) {
    memset( &__rtn_port, 0, sizeof(SNAME));
    recv( &__parent_uport, &__rtn_port, rarg, &rres );
    /* Find which child this is from - for ordering. */
    for( __array_idx = 1, __uport_cnt = 0; __uport_cnt < num_children; __uport_cnt++, __array_idx++ )
        if( memcmp( &__rtn_port, &child_uports[__uport_cnt], sizeof(SNAME)) == 0 )
            break;
    /* then use __array_idx in the loop below */
    ptr = rarg;
    memcpy(&j, ptr->ra_buffer, ptr->ra_size);
    ptr = ptr->ra_next;
    /* EXTRACT OUT THE STUFF WE WANT FOR A RESERVED OBJECT HERE - FOR A PARTICULAR CHILD! */
    memcpy(&(glob.input1[__array_idx]), ptr->ra_buffer + (__C1 + 1) * sizeof(int) * __array_idx, (__C1 + 1) * sizeof(int));
    ptr = ptr->ra_next;
    memcpy(&i, ptr->ra_buffer, ptr->ra_size);
}

/* Synchronise */
num_children = (__R1 - 1 + 1);
child_stats = (uint32_t *)malloc(sizeof(uint32_t) * num_children);
returned_child_psns = (SNAME *)malloc(sizeof(SNAME) * num_children);
proc_nwait( num_children, child_stats, returned_child_psns );

num_children = (__R2 - 1 + 1);
child_psns = (SNAME *)malloc(sizeof(SNAME) * num_children);
child_uports = (SNAME *)malloc(sizeof(SNAME) * num_children);
if( proc_ncreate( "\home\chrismc\SLVS\slave_1043051400_02.exe", num_children, child_psns, child_uports ) < 0 )
    { NUCprintf("matrix: ERROR - process create failed\n");
      proc_terminate(0, (SNAME *)0);
    }

/* Argument sending bit */

SE_ARG *sarg, *ptr;
SRESULTS sres;
int __cnt, __i_cnt;

sarg = ptr = (SE_ARG *)malloc(sizeof(SE_ARG));
ptr->sa_options = OP_MESSAGE;
ptr->sa_next = (SE_ARG *)0x0;
ptr->sa_buffer = (char *)&j;
ptr->sa_size = sizeof(int);
if( !ptr->sa_next ) {
Appendix C - Message Passing Parent Example

```c
ptr->sa_next = (SE_ARG *)malloc(sizeof(SE_ARG));
ptr = ptr->sa_next;

ptr->sa_buffer = (char *)&(glob.input2);
ptr->sa_size = sizeof(glob.input2);
if( !ptr->sa_next ) {
    ptr->sa_next = (SE_ARG *)malloc(sizeof(SE_ARG));
    ptr = ptr->sa_next;
}
ptr->sa_buffer = (char *)&i;
ptr->sa_size = sizeof(int);
if( sarg->sa_next ) {
    sarg->sa_options |= OP_MULTIBUF;
}
__i_cnt = 1;
for( __cnt = 0; __cnt < num_children; __cnt++, __i_cnt++ ) {
    /* Set the index of the loop, this is pointed to by the 
     * assignment above. */
    i = __i_cnt;
    send( &child_uports[__cnt], &__parent_uport, sarg, 
    &sres );
}
/* Argument receiving bit */
{
    RE_ARG *rarg, *ptr;
    RRESULTS rres;
    SNAME __rtn_port;
    int __cnt, __array_idx, __uport_cnt;
    /* malloc() off memory to receive into. This portion of memory 
     ** needs to be large enough to hold the complete values sent 
     ** back, which is then dissected into the actual components that 
     ** the particular child calculated/changed. */
    ptr = rarg = (RE_ARG *)malloc(sizeof(RE_ARG));
    ptr->ra_options = OP_MESSAGE;
    ptr->ra_next = (RE_ARG *)0x0;
    ptr->ra_buffer = (char *)malloc(sizeof(int));
    ptr->ra_size = sizeof(int);
    if( !ptr->ra_next ) {
        ptr->ra_next = (RE_ARG *)malloc(sizeof(RE_ARG));
        ptr = ptr->ra_next;
    }
    /* NEED TO MAKE SURE THAT THIS IS THE CORRECT SIZE! */
    ptr->ra_buffer = (char *)malloc(sizeof(glob.input2));
    ptr->ra_size = sizeof(glob.input2);
    if( !ptr->ra_next ) {
        ptr->ra_next = (RE_ARG *)malloc(sizeof(RE_ARG));
        ptr = ptr->ra_next;
    }
    ptr->ra_buffer = (char *)malloc(sizeof(int));
    ptr->ra_size = sizeof(int);
    if( rarg->ra_next ) {
        rarg->ra_options |= OP_MULTIBUF;
    }
    for( __cnt = 0; __cnt < num_children; __cnt++ ) {
        memset( &__rtn_port, 0, sizeof(SNAME));
        recv( &__parent_uport, &__rtn_port, rarg, &rres );
        /* Find which child this is from - for ordering. */
        for( __array_idx = 1, __uport_cnt = 0; __uport_cnt <
...
num_children; __uport_cnt++, __array_idx++ )
{
    if( memcmp( &__rtn_port, &child_uports[__uport_cnt], sizeof(SNAME)) == 0 )
        break;

    /* then use __array_idx in the loop below */
    ptr = rarg;
    memcpy(&j, ptr->ra_buffer, ptr->ra_size);
    ptr = ptr->ra_next;
    /* EXTRACT OUT THE STUFF WE WANT FOR A RESERVED OBJECT HERE - FOR A PARTICULAR CHILD! */
    memcpy(&glob.input2[__array_idx], ptr->ra_buffer + (__C2 + 1) * sizeof(int) * __array_idx, (__C2 + 1) * sizeof(int));
    ptr = ptr->ra_next;
    memcpy(&i, ptr->ra_buffer, ptr->ra_size);
}

/* Synchronise */
num_children = (__R2 - 1 + 1);
child_stats = (uint32_t *)malloc(sizeof(uint32_t) * num_children);
returned_child_psns = (SNAME *)malloc(sizeof(SNAME) * num_children);
proc_nwait( num_children, child_stats, returned_child_psns );

num_children = (__R3 - 1 + 1);
child_psns = (SNAME *)malloc(sizeof(SNAME) * num_children);
child_uports = (SNAME *)malloc(sizeof(SNAME) * num_children);
if( proc_ncreate( "/home/chrismc/SLVS/slave_1043051400_03.exe", num_children, child_psns, child_uports ) < 0 ){
    NUCprintf("matrix: ERROR - process create failed\n");
    proc_terminate(0, (SNAME *)0);
}

/* Argument sending bit */
{
    SE_ARG *sarg, *ptr;
    SRESULTS sres;
    int __cnt, __i_cnt;
    sarg = ptr = (SE_ARG *)malloc(sizeof(SE_ARG));
    ptr->sa_options = OP_MESSAGE;
    ptr->sa_next = (SE_ARG *)0x0;
    ptr->sa_buffer = (char *)&j;
    ptr->sa_size = sizeof(int);
    if( !ptr->sa_next ){
        ptr->sa_next = (SE_ARG *)malloc(sizeof(SE_ARG));
        ptr = ptr->sa_next;
    }
    ptr->sa_buffer = (char *)&glob.result;
    ptr->sa_size = sizeof(glob.result);
    if( !ptr->sa_next ){
        ptr->sa_next = (SE_ARG *)malloc(sizeof(SE_ARG));
        ptr = ptr->sa_next;
    }
    ptr->sa_buffer = (char *)&i;
    ptr->sa_size = sizeof(int);
    if( sarg->sa_next ){

Appendix C - Message Passing Parent Example

sarg->sa_options |= OP_MULTIBUF;
}
__i_cnt = 1;
for(___cnt = 0; ___cnt < num_children; ___cnt++, __i_cnt++) {
    /* Set the index of the loop, this is pointed to by the 
     ** assignment above. */
    i = __i_cnt;
    send( &child_uports[___cnt], &__parent_uport, sarg,
     &sres );
}

/* Argument receiving bit */
{
    RE_ARG *rarg, *ptr;
    RRESULTS rres;
    SNAME __rtn_port;
    int __cnt, __array_idx, __uport_cnt;
    /* malloc() off memory to receive into. This portion of memory 
    ** needs to be large enough to hold the complete values sent 
    ** back, which is then dissected into the actual components that 
    ** the particular child calculated/changed. */
    ptr = rarg = (RE_ARG *)malloc(sizeof(RE_ARG));
    ptr->ra_options = OP_MESSAGE;
    ptr->ra_next = (RE_ARG *)0x0;
    ptr->ra_buffer = (char *)malloc(sizeof(int));
    ptr->ra_size = sizeof(int);
    if( !ptr->ra_next ) {
        ptr->ra_next = (RE_ARG *)malloc(sizeof(RE_ARG));
        ptr = ptr->ra_next;
    }
    /* NEED TO MAKE SURE THAT THIS IS THE CORRECT SIZE! */
    ptr->ra_buffer = (char *)malloc(sizeof(int));
    ptr->ra_size = sizeof(glob.result);
    if( !ptr->ra_next ) {
        ptr->ra_next = (RE_ARG *)malloc(sizeof(RE_ARG));
        ptr = ptr->ra_next;
    }
    memset( &__rtn_port, 0, sizeof(SNAME));
    recv( &__parent_uport, &__rtn_port, rarg, &rres );
    /* Find which child this is from - for ordering. */
    for(___array_idx = 1, __uport_cnt = 0; __uport_cnt < 
    num_children; __uport_cnt++, __array_idx++)
    {  
        if( memcmp( &__rtn_port, 
            &child_uports[__uport_cnt], sizeof(SNAME)) == 0 )
            break;
    }
    /* then use __array_idx in the loop below */
    ptr = rarg;
    memcpy(&j, ptr->ra_buffer, ptr->ra_size);
    ptr = ptr->ra_next;
    /* EXTRACT OUT THE STUFF WE WANT FOR A RESERVED OBJECT 
    HERE - FOR A PARTICULAR CHILD! */
Appendix C - Message Passing Parent Example

```c
memcpy(&glob.result[__array_idx]), ptr->ra_buffer +
(__C3 + 1) * sizeof(int) * __array_idx, (__C3 + 1) * sizeof(int));
ptr = ptr->ra_next;
memcpy(i, ptr->ra_buffer, ptr->ra_size);
}
}

/* Synchronise */
num_children = (__R3 - 1 + 1);
child_stats = (uint32_t *)malloc(sizeof(uint32_t) *
num_children);
returned_child_psns = (SNAME *)malloc(sizeof(SNAME) *
num_children);
proc_nwait( num_children, child_stats, returned_child_psns );

first_row = 1;
last_row = __R1;
NUCprintf("Initially, I start at \"", first_row,
"" and end at \", last_row);
num_children = (last_row - first_row + 1);
child_psns = (SNAME *)malloc(sizeof(SNAME) * num_children);
child_uports = (SNAME *)malloc(sizeof(SNAME) * num_children);
if( proc_ncreate("/home/chrismc/SLVS/slave_1043051400_04.exe",
num_children, child_stats, child_uports ) < 0 )
{
    NUCprintf("matrix: ERROR - process create failed\n");
    proc_terminate(0, (SNAME *)0);
}
/* Argument sending bit */

SE_ARG *sarg, *ptr;
SRESULTS sres;
int __cnt, __row_cnt;
sarg = ptr = (SE_ARG *)malloc(sizeof(SE_ARG));
ptr->sa_options = OP_MESSAGE;
ptr->sa_next = (SE_ARG *)0x0;
ptr->sa_buffer = (char *)&col;
ptr->sa_size = sizeof(int);
if( !ptr->sa_next )
    ptr->sa_next = (SE_ARG *)malloc(sizeof(SE_ARG));
ptr = ptr->sa_next;

ptr->sa_buffer = (char *)&sum;
ptr->sa_size = sizeof(int);
if( !ptr->sa_next )
    ptr->sa_next = (SE_ARG *)malloc(sizeof(SE_ARG));
ptr = ptr->sa_next;

ptr->sa_buffer = (char *)&i;
ptr->sa_size = sizeof(int);
if( !ptr->sa_next )
    ptr->sa_next = (SE_ARG *)malloc(sizeof(SE_ARG));
ptr = ptr->sa_next;

ptr->sa_buffer = (char *)&(glob.input1);
ptr->sa_size = sizeof(glob.input1);
if( !ptr->sa_next )
    ptr->sa_next = (SE_ARG *)malloc(sizeof(SE_ARG));
ptr = ptr->sa_next;
```

Appendix C - Message Passing Parent Example

```c
ptr->sa_buffer = (char *)&row;
ptr->sa_size = sizeof(int);
if( !ptr->sa_next ) {
    ptr->sa_next = (SE_ARG *)malloc(sizeof(SE_ARG));
    ptr = ptr->sa_next;
}
ptr->sa_buffer = (char *)&(glob.input2);
ptr->sa_size = sizeof(glob.input2);
if( !ptr->sa_next ) {
    ptr->sa_next = (SE_ARG *)malloc(sizeof(SE_ARG));
    ptr = ptr->sa_next;
}
ptr->sa_buffer = (char *)&(glob.result);
ptr->sa_size = sizeof(glob.result);
if( !ptr->sa_next ) {
    ptr->sa_next = (SE_ARG *)malloc(sizeof(SE_ARG));
    ptr = ptr->sa_next;
}
__row_cnt = first_row;
for( __cnt = 0; __cnt < num_children; __cnt++, __row_cnt++ )
{
    /* Set the index of the loop, this is pointed to by the 
     * assignment above. */
    row = __row_cnt;
    send( &child_uports[__cnt], &__parent_uport, sarg,
    &sres);
}
/* Argument receiving bit */
{
    RE_ARG *rarg, *ptr;
    RRESULTS rres;
    SNAME __rtn_port;
    int __cnt, __array_idx, __uport_cnt;
    /* malloc() off memory to receive into. This portion of memory 
     * needs to be large enough to hold the complete values sent 
     * back, which is then dissected into the actual components that 
     * the particular child calculated/changed. */
    ptr = rarg = (RE_ARG *)malloc(sizeof(RE_ARG));
    ptr->ra_options = OP_MESSAGE;
    ptr->ra_next = (RE_ARG *)0x0;
    ptr->ra_buffer = (char *)malloc(sizeof(int));
    ptr->ra_size = sizeof(int);
    if( !ptr->ra_next ) {
        ptr->ra_next = (RE_ARG *)malloc(sizeof(RE_ARG));
        ptr = ptr->ra_next;
    }
    ptr->ra_buffer = (char *)malloc(sizeof(int));
    ptr->ra_size = sizeof(int);
    if( !ptr->ra_next ) {
        ptr->ra_next = (RE_ARG *)malloc(sizeof(RE_ARG));
        ptr = ptr->ra_next;
    }
    ptr->ra_buffer = (char *)malloc(sizeof(int));
    ptr->ra_size = sizeof(int);
    if( !ptr->ra_next ) {
        ptr->ra_next = (RE_ARG *)malloc(sizeof(RE_ARG));
        ptr = ptr->ra_next;
    }
}
```

/* NEED TO MAKE SURE THAT THIS IS THE CORRECT SIZE! */
ptr->ra_buffer = (char *)malloc(sizeof(glob.input1));
ptr->ra_size = sizeof(glob.input1);
if( !ptr->ra_next ) {
    ptr->ra_next = (RE_ARG *)malloc(sizeof(RE_ARG));
    ptr = ptr->ra_next;
}
ptr->ra_buffer = (char *)malloc(sizeof(int));
ptr->ra_size = sizeof(int);
if( !ptr->ra_next ) {
    ptr->ra_next = (RE_ARG *)malloc(sizeof(RE_ARG));
    ptr = ptr->ra_next;
}
/* NEED TO MAKE SURE THAT THIS IS THE CORRECT SIZE! */
ptr->ra_buffer = (char *)malloc(sizeof(glob.input2));
ptr->ra_size = sizeof(glob.input2);
if( !ptr->ra_next ) {
    ptr->ra_next = (RE_ARG *)malloc(sizeof(RE_ARG));
    ptr = ptr->ra_next;
}
/* NEED TO MAKE SURE THAT THIS IS THE CORRECT SIZE! */
ptr->ra_buffer = (char *)malloc(sizeof(glob.result));
ptr->ra_size = sizeof(glob.result);
if( rarg->ra_next ) {
    rarg->ra_options |= OP_MULTIBUF;
}
for( __cnt = 0; __cnt < num_children; __cnt++ ) {
    memset( &__rtn_port, 0, sizeof(SNAME));
    recv( &__parent_uport, &__rtn_port, rarg, &rres );
    /* Find which child this is from - for ordering. */
    for( __array_idx = first_row, __uport_cnt = 0;
        __uport_cnt < num_children; __uport_cnt++, __array_idx++ )
        if( memcmp( &__rtn_port, &child_uports[__uport_cnt], sizeof(SNAME)) == 0 )
            break;
    /* then use __array_idx in the loop below */
    ptr = rarg;
    memcpy(&col, ptr->ra_buffer, ptr->ra_size);
    ptr = ptr->ra_next;
    memcpy(&sum, ptr->ra_buffer, ptr->ra_size);
    ptr = ptr->ra_next;
    memcpy(&i, ptr->ra_buffer, ptr->ra_size);
    ptr = ptr->ra_next;
    /* EXTRACT OUT THE STUFF WE WANT FOR A RESERVED OBJECT HERE - FOR A PARTICULAR CHILD! */
    memcpy(&glob.input1[__array_idx], ptr->ra_buffer +
        (__C1 + 1) * sizeof(int) * __array_idx, (__C1 + 1) * sizeof(int));
    ptr = ptr->ra_next;
    memcpy(&row, ptr->ra_buffer, ptr->ra_size);
    ptr = ptr->ra_next;
    /* EXTRACT OUT THE STUFF WE WANT FOR A RESERVED OBJECT HERE - FOR A PARTICULAR CHILD! */
    memcpy(&glob.input2[__array_idx], ptr->ra_buffer +
        (__C2 + 1) * sizeof(int) * __array_idx, (__C2 + 1) * sizeof(int));
    ptr = ptr->ra_next;
    /* EXTRACT OUT THE STUFF WE WANT FOR A RESERVED OBJECT HERE - FOR A PARTICULAR CHILD! */
memcpy(&(glob.result[__array_idx]), ptr->ra_buffer + 
(C3 + 1) * sizeof(int) * __array_idx, (C3 + 1) * sizeof(int));
}

/* Synchronise */
num_children = (last_row - first_row + 1);
child_stats = (uint32_t *)malloc(sizeof(uint32_t) * 
num_children);
returned_child_psns = (SNAME *)malloc(sizeof(SNAME) * 
num_children);
proc_nwait( num_children, child_stats, returned_child_psns );

NUCprintf("\n", "Finished");
/* End time stamp */
get_up_time( &stop_time );
diff_time = sub_time(&start_time, &stop_time);
NUCprintf("Total time == %d.%d\n", diff_time.tv_sec, 
diff_time.tv_usec);
{

/* Store the results in a file for performance data. */
int32_t ofd;
SNAME psn;
char buf[32];
get_psn(&psn);
sprintf(buf, "%dx%d:%d:%d.%d:<%d,0x%x>\n", __R1, __C1, 
num_hosts, diff_time.tv_sec, diff_time.tv_usec, psn.sn_object, 
psn.sn_origin);
ofd = ropen("/home/chrismc/matrix.dat", O_CREAT | O_WRONLY |
O_APPEND, 0660);
fwrite(ofd, buf, strlen(buf));
fclose(ofd);
}
proc_terminate(0, (SNAME *)0);
return(0);
Appendix D  Message Passing Child Example

slave_159395400_04.c

/* Include Files */
#include <sname.h>
#include <portname.h>
#include <ipc.h>
#include <rhodos.h>
#include <time.h>
#include "matrix.h"

void main(int argc, char **argv)
{
    int32_t num_children;
    uint32_t *child_stats;
    SNAME *returned_child_psns;
    SNAME *child_psns, *child_uports, child_info;
    int32_t child_return_value, child_cnt;
    SNAME __parent_port;
    get_uport( &__child_uport );
    /* Argument receiving bit */
    {
        RE_ARG *rarg, *ptr;
        RESULTS rres;
        memset(&__parent_port, 0, sizeof(SNAME));
        ptr = rarg = (RE_ARG *)malloc(sizeof(RE_ARG));
        rarg->ra_options = OP_MESSAGE;
        rarg->ra_next = (RE_ARG *)0x0;
        ptr->ra_buffer = (char *)&col;
        ptr->ra_size = sizeof(int);
        if( !ptr->ra_next ) {
            ptr->ra_next = (RE_ARG *)malloc(sizeof(RE_ARG));
            ptr = ptr->ra_next;
        }
        ptr->ra_buffer = (char *)&i;
        ptr->ra_size = sizeof(int);
        if( !ptr->ra_next ) {
            ptr->ra_next = (RE_ARG *)malloc(sizeof(RE_ARG));
            ptr = ptr->ra_next;
        }
        ptr->ra_buffer = (char *)&(glob.input1);
        ptr->ra_size = sizeof(glob.input1);
        if( !ptr->ra_next ) {
            ptr->ra_next = (RE_ARG *)malloc(sizeof(RE_ARG));
            ptr = ptr->ra_next;
        }
        ptr->ra_buffer = (char *)&(glob.input1);
        ptr->ra_size = sizeof(glob.input1);
        if( !ptr->ra_next ) {
            ptr->ra_next = (RE_ARG *)malloc(sizeof(RE_ARG));
            ptr = ptr->ra_next;
        }
    }
}
Appendix D - Message Passing Child Example

```c
ptr->ra_buffer = (char *)&row;
ptr->ra_size = sizeof(int);
if( !ptr->ra_next ) {
    ptr->ra_next = (RE_ARG *)malloc(sizeof(RE_ARG));
    ptr = ptr->ra_next;
}
ptr->ra_buffer = (char *)&(glob.input2);
ptr->ra_size = sizeof(glob.input2);
if( !ptr->ra_next ) {
    ptr->ra_next = (RE_ARG *)malloc(sizeof(RE_ARG));
    ptr = ptr->ra_next;
}
ptr->ra_buffer = (char *)&(glob.result);
ptr->ra_size = sizeof(glob.result);
if( rarg->ra_next ) {
    rarg->ra_options |= OP_MULTIBUF;
}
recv( &__child_uport, &__parent_port, rarg, &rres );
}
/* Main computation bit */
for( col = 1; col <= __C2; col++ )
{
    sum = 0;
    for( i = 1; i <= __C1; i++ )
    {
        sum = sum + glob.input1[row][i] * glob.input2[i][col];
    }
    glob.result[row][col] = sum;
}
/* Argument sending bit */
SE_ARG *sarg, *ptr;
SRESULTS sres;
ptr = sarg = (SE_ARG *)malloc(sizeof(SE_ARG));
sarg->sa_options = OP_MESSAGE;
sarg->sa_next = (SE_ARG *)0x0;
ptr->sa_buffer = (char *)&col;
ptr->sa_size = sizeof(int);
if( !ptr->sa_next ) {
    ptr->sa_next = (SE_ARG *)malloc(sizeof(SE_ARG));
    ptr = ptr->sa_next;
}
ptr->sa_buffer = (char *)&sum;
ptr->sa_size = sizeof(int);
if( !ptr->sa_next ) {
    ptr->sa_next = (SE_ARG *)malloc(sizeof(SE_ARG));
    ptr = ptr->sa_next;
}
ptr->sa_buffer = (char *)&i;
ptr->sa_size = sizeof(int);
if( !ptr->sa_next ) {
    ptr->sa_next = (SE_ARG *)malloc(sizeof(SE_ARG));
    ptr = ptr->sa_next;
}
ptr->sa_buffer = (char *)&(glob.input1);
ptr->sa_size = sizeof(glob.input1);
if( !ptr->sa_next ) {
    ptr->sa_next = (SE_ARG *)malloc(sizeof(SE_ARG));
    ptr = ptr->sa_next;
}
```
ptr = ptr->sa_next;
}
ptr->sa_buffer = (char *)&row;
ptr->sa_size = sizeof(int);
if( !ptr->sa_next ) {
    ptr->sa_next = (SE_ARG *)malloc(sizeof(SE_ARG));
    ptr = ptr->sa_next;
}
ptr->sa_buffer = (char *)&(glob.input2);
ptr->sa_size = sizeof(glob.input2);
if( !ptr->sa_next ) {
    ptr->sa_next = (SE_ARG *)malloc(sizeof(SE_ARG));
    ptr = ptr->sa_next;
}
ptr->sa_buffer = (char *)&(glob.result);
ptr->sa_size = sizeof(glob.result);
if( sarg->sa_next ) {
    sarg->sa_options |= OP_MULTIBUF;
}
send( __parent_port, __child_uport, sarg, &sres );
proc_terminate(0, (SNAME *)0);

/* END OF FILE */
matrix.h

/* Header file for the program matrix */

#ifndef _matrix_h_
#define _matrix_h_

#include <sname.h>
extern SNAME __parent_uport;
extern SNAME __child_uport;

#define __MATRIX_SIZE_ROW 2
#define __MATRIX_SIZE_COL 5
#define __R1 2
#define __R2 5
#define __R3 2
#define __C1 5
#define __C2 2
#define __C3 2

typedef struct globalmemory {
  int input1[(__R1 + 1)[__C1 + 1];
  int input2[(__R2 + 1)[__C2 + 1];
  int result[(__R3 + 1)[__C3 + 1];
} globalmemory;
extern globalmemory glob;
extern int row;
extern int col;
extern int i;
extern int j;
extern int sum;
extern int first_row;
extern int last_row;

#endif /* _matrix_h_ */

/* END OF FILE */
matrix_var.c

/* Global variables for the program matrix */

#include "matrix.h"

SNAME __parent_uport;
SNAME __child_uport;
globalmemory glob;
int row;
int col;
int i;
int j;
int sum;
int first_row;
int last_row;

/* END OF FILE */
Appendix F  Shared Memory Parent Example

matrix.c

/* Include Files */
#include <sname.h>
#include <portname.h>
#include <ipc.h>
#include <rhodos.h>
#include <time.h>
#include <rufs/iface.h>
#include <rufs/file_cntl.h>
#include "matrix.h"
#include "dsm.h"

timeval sub_time(timeval *, timeval *);

int main( int argc, char *argv[] )
{
    int32_t num_children;
    uint32_t *child_stats;
    SNAME *returned_child_psns;
    SNAME __dsm_sp_name;

    /* Needed for testing */
    int32_t num_hosts;

timeval start_time, stop_time, diff_time;

    /* Execution environment bug fix */
    int *z_z;
    z_z = (int *)malloc(sizeof(z_z));

    /* Performance skeleton */
    if( argc == 2 )
        num_hosts = atoi(argv[1]);
    else
        num_hosts = 999999;

    __dsm_num_sems = DSM_NUM_SEMS;
    __dsm_num_barriers = DSM_NUM_BARRIERS;

    get_uport( &__parent_uport );

    /* Start time stamp */
    get_up_time( &start_time );

    /* Matrix initialisation */
    for( i = 0; i <= __R1; i++ )
    {
        for( j = 0; j <= __C1; j++ )
        {
            glob.input1[i][j] = 1;
for( i = 0; i <= __R2; i++ )
{
    for( j = 0; j <= __C2; j++ )
    {
        glob.input2[i][j] = 2;
    }
}
for( i = 0; i <= __R3; i++ )
{
    for( j = 0; j <= __C3; j++ )
    {
        glob.result[i][j] = 0;
    }
}

first_row = 1;
last_row = __R1;
NUCprintf("Initially, I start at ", first_row,
" and end at ", last_row);
num_children = (last_row - first_row + 1);
glob.start_row[0] = 0;
glob.start_row[1] = 1;
glob.start_row[2] = 2;
glob.start_row[3] = 3;
glob.start_row[4] = 4;
glob.start_row[5] = 5;

dsm_1043051400_01_mem = (struct __dsm_1043051400_01_mem
*)start_dsm_args("/home/chrismc/SLVS/slave_1043051400_01.exe",
&_dsm_sp_name, RELEASE, sizeof(struct __dsm_1043051400_01_mem),
&num_children, __dsm_sem, __dsm_barrier, __dsm_num_sems,
__dsm_num_barriers);


*/ Populate the dsm_1043051400_01_mem pointer here, so that
** the slave process(es) get the correct starting values. */
/*#######*/
dsm_1043051400_01_mem->col = col;
dsm_1043051400_01_mem->sum = sum;
dsm_1043051400_01_mem->i = i;
memcpy(&dsm_1043051400_01_mem->glob), &glob, sizeof(struct
globalmemory));
dsm_1043051400_01_mem->row = row;
/*#######*/

/* dsm structure all complete now */
dsm_barrier(__dsm_barrier[0]);

/* Synchronise together to let parent get data out of the dsm
structure */
dsm_barrier(__dsm_barrier[1]);

/* extract DSM stuff from dsm_1043051400_01_mem here */
/*#######*/
col = dsm_1043051400_01_mem->col;
sum = dsm_1043051400_01_mem->sum;
i = dsm_1043051400_01_mem->i;
memcpy(&glob, &(dsm_1043051400_01_mem->glob), sizeof(struct globalmemory));
row = dsm_1043051400_01_mem->row;
/*###*/
/* dsm structure deconstructed now */

num_children = (last_row - first_row + 1);
child_stats = (uint32_t *)malloc(sizeof(uint32_t) * num_children);
returned_child_psns = (SNAME *)malloc(sizeof(SNAME) * num_children);
proc_nwait(num_children, child_stats, returned_child_psns);
NUCprintf("%s\n", "Finished");

/* End time stamp */
get_up_time(&stop_time);
diff_time = sub_time(&start_time, &stop_time);
NUCprintf("Total time == %d.%d\n", diff_time.tv_sec, diff_time.tv_usec);

/* Clean up DSM barriers, locks, and other stuff before exiting.*/
end_dsm();

{ /* store the results in a file */
int32_t ofd;
SNAME psn;
char buf[32];
get_psn(&psn);
sprintf(buf, "%dx%d:%d:%d.%d:<%d,0x%x>\n", __R1, __C1, num_hosts, diff_time.tv_sec, diff_time.tv_usec, psn.sn_object, psn.sn_origin);
ofd = ropen("/home/chrismc/matrix.dat", O_CREAT | O_WRONLY | O_APPEND, 0660);
rwrite(ofd, buf, strlen(buf));
rclose(ofd);
}
proc_terminate(0, (SNAME *)0);
return(0);}
Appendix G  Shared Memory Child Example

slave_159395400_01.c

/* Include Files */
#include <sname.h>
#include <portname.h>
#include <ipc.h>
#include <rhodos.h>
#include <time.h>
#include "matrix.h"
#include "dsm.h"

void main(int argc, char **argv)
{
    int32_t num_children;
    uint32_t *child_stats;
    SNAME *returned_child_psns;
    SNAME ___dsm_sp_name;

    ___dsm_num_sems = atoi(argv[0]);
    ___dsm_num_barriers = atoi(argv[1]);

    dsm_22839_mem = (struct __dsm_22839_mem *)dsm_parstart(&___dsm_slave_num, &___dsm_sp_name, &___dsm_num_procs,
___dsm_num_sems, ___dsm_num_barriers, ___dsm_sem, ___dsm_barrier);

    /* Start together */
    dsm_barrier(___dsm_barrier[0]);

    row = dsm_22839_mem->glob.start_row[___dsm_slave_num];
    for( col = 1; col <= ___C2; col++ )
    {
        sum = 0;
        for( i = 1; i <= ___C1; i++ )
        {
            sum = sum + dsm_22839_mem->glob.input1[row][i] *
        dsm_22839_mem->glob.input2[i][col];
        }

        /* Synchronise here */
        lock(___dsm_sem[1]);
        dsm_22839_mem->glob.result[row][col] = sum;
        unlock(___dsm_sem[1]);
    }

    /* Synchronise together to let parent get data out of the
** dsm structure */
    dsm_barrier(___dsm_barrier[1]);
    proc_terminate(0, (SNAME *)0);
}

/* END OF FILE */
Appendix H  Shared Memory
Miscellaneous
Generated Files

matrix.h

/* Header file for the program matrix */

#ifndef _matrix_h_
define _matrix_h_

#include <sname.h>
extern SNAME __parent_uport;
extern SNAME __child_uport;

#define __MATRIX_SIZE_ROW 2
#define __MATRIX_SIZE_COL 5
#define __R1 5
#define __R2 10
#define __R3 5
#define __C1 10
#define __C2 5
#define __C3 5
#define max(a,b) (a > b ? a : b)
typedef struct globalmemory {
    int start_row[max((__R1 + 1), (__R2 + 1))];
    int input1[__R1 + 1][__C1 + 1];
    int input2[__R2 + 1][__C2 + 1];
    int result[__R3 + 1][__C3 + 1];
} globalmemory;
extern globalmemory glob;
extern int row;
extern int col;
extern int i;
extern int j;
extern int sum;
extern int first_row;
extern int last_row;

#endif /* _matrix_h_ */

/* END OF FILE */
matrix_var.c

/* Global variables for the program matrix */

#include "matrix.h"

SNAME __parent_uport;
SNAME __child_uport;
globalmemory glob;
int row;
int col;
int i;
int j;
int sum;
int first_row;
int last_row;

/* END OF FILE */
# Appendix H - Shared Memory Miscellaneous Generated Files

`dsm.h`

/* DSM include files */

#ifndef _dsm_h_
#define _dsm_h_

#include <space/defs.h>

#define DSM_NUM_SEMS 2
#define DSM_NUM_BARRIERS 2
SNAME __dsm_sem[DSM_NUM_SEMS],
    __dsm_barrier[DSM_NUM_BARRIERS];
int __dsm_num_sems, __dsm_num_barriers;
uint32_t __dsm_slave_num, __dsm_num_procs;

#include "dsm_1043051400_01.h"

#endif /* _dsm_h_* */

/* END OF FILE */
dsm_1043051400_01.h

/* DSM structures */

#ifndef _dsm_1043051400_01_h_
#define _dsm_1043051400_01_h_

struct __dsm_1043051400_01_mem
{
    int col;
    int sum;
    int i;
    globalmemory glob;
    int row;
};

struct __dsm_1043051400_01_mem *dsm_1043051400_01_mem;

#endif /* _dsm_1043051400_01_h_ */

/* END OF FILE */