

http://researchcommons.waikato.ac.nz/

Research Commons at the University of Waikato

Copyright Statement:

The digital copy of this thesis is protected by the Copyright Act 1994 (New Zealand).

The thesis may be consulted by you, provided you comply with the provisions of the Act and the following conditions of use:

- Any use you make of these documents or images must be for research or private study purposes only, and you may not make them available to any other person.
- Authors control the copyright of their thesis. You will recognise the author's right to be identified as the author of the thesis, and due acknowledgement will be made to the author where appropriate.
- You will obtain the author's permission before publishing any material from the thesis.

Scheme^{n}: Design, Implementation and Application

A thesis presented to the
University of Waikato
in fulfillment of the thesis requirement
for the degree of

Doctor of Philosophy

by

Wayne Conrad Schou



University of Waikato
29 May 1995

Abstract

This thesis presents the design, implementation and application of Scheme^{n}, a parallel extension of the Scheme programming language. Scheme^{n} has been designed for a distributed memory architecture to take advantage of explicitly specified large and medium grain parallelism. The language extensions and support provide for both data and control parallelism and the use of speculative computing to maximise parallel activity. Support is also provided for explicit process control for the assignment and termination of concurrent tasks.

The main mechanism for introducing parallelism into a Scheme^{n} program is the future construct originally defined by Halstead for use in the Multilisp language and used in other parallel lisp designs. Implicit futures, which are dynamically assigned to available processes, via a distributed load balancing mechanism, have been used to hide the evaluation mechanism from the algorithm design.

The thesis introduces the concepts of distributed processing. From these, the design of the Scheme^{n} language extensions and the supporting mechanisms required for communication between the concurrent tasks is given. An implementation of Scheme^{n} on a network of multiprocessors is then described. The behaviour of the implementation is analysed for a variety of communication topologies and the overhead of parallel invocation is quantified.

Finally, an application of the language to the factorisation of large integers is demonstrated. Factorisation of an integer into its component prime factors is a computationally intensive task and several different methods exist for determining the factors and their primality. The factorisation process is parallelised by applying several factorisation methods to an integer concurrently. Where possible, the parallel features of a particular method are also utilised. A description is given of the factorisation methods and how they are combined. The application's performance is analysed to provide understanding of the behaviour of the Scheme^{n} system.

Acknowledgements

First and foremost I would like to thank my supervisor, Associate Professor Kevin Broughan, for his guidance and support; for knowing what's important and keeping me on track. I would also like to thank Professor Ian Graham for setting up the parallel processing project and for his efforts to improve and maintain the hardware and software. I would also like to acknowledge the camaraderie of my fellow graduate students; without whom life would have been dull. Special thanks to Doctor Diane Koorey Willcock for leading the way into the practicalities of parallel processing and Steven Galbraith for his serial implementation of the factorisation package. And finally, I would like to acknowledge the support of the Department of Mathematics and Statistics in providing the facilities and environment for research, and the financial support of the University Grants Committee for their postgraduate scholarship.

Contents

List of Figures v

List of Tables vii

1 Introduction 1
- 1.1 Introduction . . . 1
- 1.2 Improving Performance . . . 1
- 1.3 Parallel Processing . . . 4
  - 1.3.1 Parallel Algorithms . . . 6
- 1.4 Concurrent Language Design Issues . . . 7
- 1.5 Concurrent Programming Languages . . . 9
- 1.6 The Thesis Question . . . 10
- 1.7 Scheme^{n} . . . 13
- 1.8 Parallel Lisps . . . 14
- 1.9 Thesis Structure . . . 16

2 Distributed Processing Concepts 18
- 2.1 Introduction . . . 18
- 2.2 Communication . . . 18
  - 2.2.1 Message Passing . . . 19
  - 2.2.2 Message Passing Environments . . . 21
- 2.3 Synchronisation . . . 23
- 2.4 Shared Variables . . . 24
  - 2.4.1 Serializability . . . 24
  - 2.4.2 Emulating Shared Memory . . . 25
- 2.5 Topology . . . 27
- 2.6 Partitioning, Scheduling and Allocation . . . 29
  - 2.6.1 Program Partitioning . . . 31
  - 2.6.2 Task Scheduling . . . 31
  - 2.6.3 Task Allocation . . . 32
- 2.7 Methods of Introducing Parallelism . . . 33
- 2.8 Futures . . . 35
  - 2.8.1 Future Concept . . . 35
  - 2.8.2 Future Types . . . 37

3 Scheme^{n} - A Distributed Language Design 38
- 3.1 Introduction . . . 38
- 3.2 Scheme^{n} Parallel Constructs . . . 38

3.2.1 `remote` . . . 39
3.2.2 `touch` . . . 39
3.2.3 `future?` . . . 40
3.2.4 `lazy` . . . 40
3.2.5 `force` . . . 42
3.2.6 `make-semaphore` . . . 42
3.2.7 `request` . . . 42
3.2.8 `signal` . . . 42
3.2.9 `make-shared` . . . 43
3.2.10 `parallel` . . . 44
3.2.11 `wait` . . . 46
3.2.12 `kill` . . . 47
3.2.13 `kill-with-no-value` . . . 47
3.2.14 `on-kill` . . . 48
3.3 Examples . . . 48
3.3.1 Example 1: Fibonacci Numbers . . . 48
3.3.2 Example 2: Finding Eigenvalues . . . 50
3.3.3 Example 3: Parallel Vector/Matrix Addition . . . 51
3.3.4 Example 4: Streams . . . 52
3.3.5 Example 5: Vector Dot Product . . . 53

4 Distributed System Design **55**
4.1 Introduction . . . 55
4.2 Scheme^n Design Choices . . . 55
4.3 Load . . . 58
4.3.1 Load Balancing . . . 59
4.3.2 Load Estimates . . . 61
4.4 Future Structure . . . 63
4.4.1 Future Table . . . 70
4.4.2 Future Checking . . . 71
4.4.3 Future Garbage Collection . . . 72
4.5 Explicit Termination . . . 73
4.5.1 The Kill Message . . . 74

5 Implementation **80**
5.1 Introduction . . . 80
5.2 Implementation Environment . . . 80
5.2.1 The Processor: Transputer . . . 81
5.2.2 The Operating System: Helios . . . 81
5.2.3 The Base Language: Scheme->C . . . 82
5.3 Messages . . . 82
5.3.1 Transmitting Scheme^n Data . . . 84
5.3.2 Maintaining Structure Sharing . . . 88
5.4 Communication Requirements . . . 90
5.5 Remote Initiation . . . 93
5.6 The Remote Handler . . . 94
5.7 Remote Evaluation . . . 95
5.8 Input and Output Servers . . . 96

5.8.1 Output Server 96
5.8.2 Input Server 97
5.9 The Future Handler 98

6 Analysis **100**
6.1 Introduction 100
6.2 Remote Overhead 103
6.3 Communication Overhead 105
6.4 Implicit Future Checking 110
6.5 Load Balancing 111

7 Application: Factorisation **120**
7.1 Introduction 120
7.2 Integer Factorisation Methods 121
7.2.1 Trial Division 121
7.2.2 Pollard's $\rho - 1$ Method 123
7.2.3 Morrison and Brillhart's Continued Fraction Method 125
7.2.4 Other Methods 128
7.3 Primality 129
7.3.1 Primality Testing 130
7.3.2 Primality Proving 131
7.4 A Parallel Factorisation Package 132
7.4.1 Repeated Factors 134
7.4.2 Perfect Powers (when $N = q^{\alpha}$) 135
7.5 Analysis 136
7.5.1 Parallel vs Sequential 136
7.5.2 Proving vs Testing 137
7.5.3 Parallel Continued Fraction Method 139
7.5.4 Effect of Process Configuration on Load Balancing 142

8 Summary, Conclusions and Further Work **151**
8.1 Summary 151
8.2 Conclusions 152
8.3 Further Work 154

Bibliography **156**

List of Figures

2.1 Serializability 24
2.2 Serializability between multiple variables 25

3.1 Example 1: Scheme^n code for generating the nth Fibonacci number . 49
3.2 Example2: Scheme^n code for finding eigenvalues. 50
3.3 Example 3: Parallel Vector/Matrix Addition 51
3.4 Example 4: Stream 52
3.5 Example 5: Vector Dot Product 53
3.6 Example 5: Unordered Vector Dot Product 53
3.7 Example 5: Stream Vector Dot Product 54

5.1 Scheme^n list representation of ((a) b) 85
5.2 Contiguous representation of ((a) b) 85
5.3 Prefix representation of ((a) b) 86
5.4 Prefix representation with list length of ((a) b) 86
5.5 LWP configuration 92

6.1 The Physical Transputer Configuration 101
6.2 Fully Connected Topology 101
6.3 Ring Topology 102
6.4 Binary Tree Topology 102
6.5 Star Topology 102
6.6 Grid Topology 102
6.7 The Effect of Overhead 104
6.8 Remote Overhead Measurement Code 105
6.9 Communication Performance - I 107
6.10 Communication Performance - II 109
6.11 Code to demonstrate load balancing 113
6.12 Load Balancing Sequence I 114
6.13 Load Balancing Sequence II 115
6.14 Balancing for tree configuration 118

7.1 Pseudo code of top-level factorisation routine 133
7.2 Parallel evaluation of Morrison and Brillhart's Continued Fraction Method 140
7.3 Pattern of remote task creation during factorisation 142
7.4 Process loads for a fully connected topology 144
7.5 Process loads for a grid topology 145

7.6 Process loads for a ring topology 146
7.7 Process loads for a tree topology 147

List of Tables

6.1 Effect of Implicit Future Checking 111
6.2 Load Estimates I 115
6.3 Load Estimates II 115
6.4 Connection Matrix 119

7.1 Comparison of factorisation times using primality testing and proving 137
7.2 Comparison of factorisation times using primality testing and proving 138
7.3 Parallel Continued Fraction Performance for $10^{40} + 5$ 141
7.4 Example factorisation times for 40 digit numbers 141
7.5 Evaluation and idle times for different configurations 150

Chapter 1

Introduction

1.1 Introduction

This thesis introduces Scheme^{n}, a parallel extension of the Scheme programming language.

This chapter introduces the topics of parallel processing and parallel language design. A simple rationale for the current interest in parallel processing and a brief summary of the variety of parallel methods available in existing parallel environments are given. The main design issues of a concurrent language are then itemised. From the field of parallel language design, the question to be answered by this thesis is presented. An outline of the resulting parallel language, Scheme^{n}, is then given. A survey of other existing parallel Lisp and Scheme implementations is presented, highlighting their similarities and differences from Scheme^{n}. The chapter concludes with an overview of the thesis structure.

1.2 Improving Performance

To solve a problem using a computing device requires the problem to be specified as an *algorithm*. An algorithm is a set of instructions, or *basic operations*, which describe the process required to arrive at a solution [42]. When applied to a computing device these basic operations are evaluated in a particular order to produce a solution to the relevant problem. A goal of computing is to perform the requested operations as quickly as possible. To reduce the execution time of a problem we could:

- use a faster algorithm,

- reduce the execution time required for each basic operation or
- overlap the evaluation of operations.

Each of these methods can be applied to the computing problem separately or combined together for cumulative improvements. The first method involves changes to the software only whereas the second and third methods described above require support from the computing hardware.

Changing the algorithm used to solve a given problem to one that requires fewer or simpler basic operations will reduce the execution time. The algorithm which has the best performance will depend on the computing hardware used to evaluate it. For any given problem there will be a lower bound to the number of basic operations that are required to find a solution, assuming a fixed set of basic operations can be used. This lower bound gives a limit to the speed improvements that can be achieved through the selection of different algorithms. Once a fastest algorithm has been chosen no further significant performance improvements can be made using this method for a given computing device.

Changes to an algorithm will only affect the execution time of that particular algorithm and have no effect on other algorithms executed using the same hardware. Improvements to execution speed for a wide range of problems will require modifications to be made to each algorithm used.

Given an algorithm which specifies a set of basic operations, the execution time of the algorithm can also be reduced by using a computing device which can evaluate these basic operations faster. Computer hardware modifications include increasing the clock speed of the central processing unit (CPU), which increases the speed at which the basic operations are processed, adding a floating point coprocessor to improve the performance of floating point operations, or using memory with a faster access time (instruction and data caches in particular) [33]. Each of these improvements can make significant reductions to evaluation times. In theory, however, there are physical limits to the speed at which a single computer unit can receive and process data.

In practice, there is a more severe limit on hardware improvements due to the financial costs involved in the research, development and manufacture of higher quality components. At some point the costs of enhancements to the hardware will outweigh the matching improvements in algorithm evaluation speeds for most applications.

The third method of reducing the evaluation time of a given problem considers the order of basic operation evaluation. Given a set of basic operations to be evaluated it is necessary to perform some operations in a specific order because of data dependencies. A *data dependency* is a relationship that exists between objects, where the evaluation of one object is dependent on the result of another. For example, in the evaluation of $3 * (2 + 1)$ the multiplication operation is data-dependent on the addition operation and cannot proceed until the addition has completed. A data dependency graph can be constructed for an algorithm which will graphically display the necessary ordering between the basic operations as a network of object nodes and directed data dependency arcs [67].

The preceding performance improvement methods have assumed that the basic operations are performed in a specified sequence, with the output of earlier operations being used as input to later operations, culminating in a final answer. The sequence of basic operations that are evaluated, however, need not be uniquely defined. When two or more operations are to be executed and their evaluation is independent of each other then the operations need not be done in any particular order but can be performed simultaneously by overlapping their execution.

Consider the time needed to sequentially evaluate an algorithm consisting of N basic operations where the result of each operation is necessary for correct evaluation of the algorithm and N is typically very large. If these basic operations are performed one-at-a-time then the time to evaluate the algorithm is $\sum_1^N t_i$, where t_i is the time needed to evaluate the i th operation. If it is further assumed that each basic operation requires the same execution time, t, then the sequential evaluation requires Nt time units to complete. If all operations could be evaluated at once the time could be reduced to just t time units, resulting in a dramatic N-fold speed-up. Although a complete overlap of all operations is usually impractical, by overlapping the execution of those operations that can be evaluated together, the time required to execute the algorithm is no longer a sum of basic operation time units but a fraction of this, dependent on the degree of overlap possible.

Overlapping of the basic operations can be accomplished in many ways, each requiring additional hardware to implement. At the lowest level, the CPU can overlap the atomic operations it performs to evaluate each basic operation. The types of atomic operations a CPU can overlap include: instruction fetching, memory access, register access, and Arithmetic Logic Unit (ALU) use. Overlapping can also occur within a sequence of instructions given to a CPU. Provided there are

no data dependencies between the atomic operations of a sequence of instructions several basic operations can be processed at once. However, when the values produced by instructions are required by future instructions or the future instructions themselves are conditional on operations being processed then time delays can occur. The overlapping of atomic operations within a CPU is not normally referred to as parallel computing as there is still a single stream of instructions and data which are evaluated in a sequential order, albeit in a parallel fashion.

As well as performing operations within a CPU simultaneously, if multiple CPUs were made available more basic operations could be performed in parallel. Each CPU can be given a sub-sequence of relatively independent basic operations to perform, the results of which can be collected together to form a single answer to the posed problem. The various models of parallel processing and their classifications are introduced in the next section.

1.3 Parallel Processing

The terms *parallel processing*, *distributed processing*, *concurrency* and *multiprocessing* are all associated with parallelism. Their definitions differ between researchers as they attempt to categorise the many different approaches to parallelism. *Concurrency* and *parallel processing* are perhaps the more general of these terms and can encompass any parallel activity. *Multiprocessing* is typically applied to parallel activity involving more than one computer.

Distributed processing is reserved for physically separated computers. This usually implies that each processor in a distributed system is asynchronously controlled, with its own local memory and no shared memory. Communication between processors is accomplished by sending explicit messages via some communication network such as ethernet, crossbar switches, common busses or static network topologies [33].

The distinction between these terms has been deliberately blurred in this thesis. Each term relies on vague measures and characteristics of a parallel environment and many parallel environments can fall into either several or none of these categories.

There are many other terms and classifications that can be applied to various aspects of parallelism. These can be applied to both the architecture and the software models used. Because the terms used are not precise in their definitions,

many classifications overlap and can even be considered as synonymous with very few exceptions.

A commonly used classification scheme is Flynn's Taxonomy [59] which partitions all computer architectures into one of four categories (SISD, SIMD, MISD, MIMD) based on the number of instruction and data streams used. Most serial computers fall into the Single-Instruction, Single Data (SISD) category, while multiprocessors can be classed as Single-Instruction, Multiple-Data (SIMD), Multiple-Instruction, Single Data (MISD) or Multiple-Instruction, Multiple-Data (MIMD).

The SIMD class of architectures consists of *vector-* and *array-processors.* These architectures have multiple processors, usually connected in a rectangular grid, and controlled synchronously from a centralised control unit. Examples of SIMD architectures include the Goodyear Aerospace Massively Parallel Processor (MPP) [33] and Thinking Machine Corporation's CM-1 Connection Machine [2]. Each processor applies the same operation but to different data objects. For applications where the operations to be performed are largely data-independent, such as data transformations, these architectures perform well. However, for applications where the operations are data-dependent, the single flow of control available to an SIMD architecture can reduce the parallel performance significantly.

Depending on the interpretation of a single data stream, the classification of MISD architectures can either be considered as an impractical combination of capabilities or contain architectures such as *pipelined* computers. If MISD is assumed to be the application of multiple instructions to a single element of a data stream then no such architectures have been designed. If, however, the multiple instructions can be applied to different elements of a data stream then the commonly used technique of pipelining can be viewed as MISD[1]. Examples of MISD architectures include the Cray Research Cray-I [33]. Both MISD and SIMD architectures require specialised control structures for parallel computation. The specialisation does not mean they are not useful, as the more general MIMD architectures suffer from additional complexity. Indeed, many of the most practical multiprocessor systems in use today are of one of these two types.

MISD can also be termed *temporal parallelism* as the parallelism available is derived from overlapping the evaluation of consecutive elements in a stream of data. SIMD architectures involve *spatial parallelism* as distinct elements of a set are evaluated together. MIMD includes both temporal and spatial parallelism as the

[1]Without this interpretation pipeline computers fall in the category of SIMD or MIMD.

evaluation of instructions and data are allowed to overlap.

Most general-purpose multiprocessors can be classified as MIMD architectures. Each processor, in a network of processors, has its own set of instructions to evaluate and data to operate on. The independence of processors allows many different models of computation to be used. The model can be chosen to match the class of algorithms that are to be developed. Many algorithms do not contain sufficient temporal or spatial parallelism for the additional programming and hardware costs to be worthwhile. If both temporal and spatial parallelism are present the MIMD architectures can reveal more of the inherent parallelism. This allows significantly more parallelism to be utilised in a wider class of algorithms than either MISD or SIMD architectures.

1.3.1 Parallel Algorithms

The goal of parallel processing is to maximise the performance of a computation through the use of multiple processors evaluating a problem concurrently, with the result becoming available in a shorter time interval.

To develop a practical parallel application the features of the language and multiprocessor system must be considered. In particular, the degree to which parallelism can be invoked is important. Algorithms vary in the amount of inherent parallelism available. Some algorithms, such as recurrence operations, are inherently sequential, where each operation is dependent on the results of the previous operation, and must therefore be performed in a strictly sequential order. Other algorithms exhibit a high degree of parallelism where multiple operations are relatively independent and can be evaluated concurrently.

To extract good parallel performance from an algorithm a certain level (dependent on the multiprocessor system) of inherent parallelism must be present. Algorithms with little inherent parallelism can not be made more parallel just because the multiprocessor system allows for it. The algorithm may have to be redesigned or a different algorithm may have to be chosen or developed in order to increase the potential of the computation to be parallelised. Similarly, algorithms which are sufficiently parallel can not always be effectively implemented.

For the inherent parallelism in an algorithm to be implemented, its style of parallelism must be expressible in the programming language. Parallel styles include the general categories of data and control parallelism as well as more exact attributes such as how and when parallelism is invoked and the amount of control

required over the execution of the parallel tasks. For example, to implement any speculative parallelism, the language must have control over the premature termination of parallel tasks. For the widest appeal, parallelism should be controlled dynamically but this increases the execution overhead during evaluation. Statically defined parallelism may provide better all round performance at the expense of generality. Compare the performance differences between statically defined compiled code and dynamically interpreted code.

Increasing the expressiveness of a language, by generalising existing programming constructs or by introducing new constructs, must be weighed against the effect this diversity can have on the language's performance. Each style of parallelism requires a different set of features from the language. Each feature requires that additional control procedures be implemented and these add an execution overhead to an algorithm's evaluation. Combining features will introduce further overheads needed to allow the features to coexist. No benefit is obtained if the inclusion of some rarely used or redundant feature introduces performance penalties for more common forms of parallelism.

Performance can also be limited by a lack of sufficient processors to evaluate tasks when required and the additional execution overhead needed to control the parallel tasks. Inappropriate use of too much parallelism can add so much additional overhead to the evaluation that the multiprocessor performance is severely degraded to a level even lower than the performance of a single processor system. A system can be overloaded to the extent that no useful work is performed as the system is forced to deal with the management of the tasks rather than their evaluation.

The level of parallelism that can be successfully implemented is therefore determined by the attributes of the multiprocessor system, the algorithm and of the programming language's ability to express the parallelism. The aim of a good language design is to make use of all possible parallel activity of a wide range of algorithms within the limits of the multiprocessor system.

1.4 Concurrent Language Design Issues

To design a general-purpose distributed programming language requires a number of decisions to be made on a variety of design issues. The design of a concurrent language is dependent on the hardware and software environment in which the language is expected to perform. Each design issue should be supported by the

environment so it can be easily implemented and provide adequate performance. The language attributes must also work together effectively, enhancing the features of each to form a consistent, coherent and complete language design. When considering the advantages and disadvantages of each design criterion the effect of one choice on the rest of the design needs to be examined. In choosing one particular feature for the design many other useful features that cannot coexist may be automatically eliminated. A balance is required among the design criteria. A system can be "bottle-necked" by a poorly chosen feature so that much of the included functionality of other features cannot be used effectively. The trade-off between design choices should therefore be considered in the context of the performance and functionality of the language as a whole.

P.D. Stotts [72] provides a summary of design issues that must be resolved when designing a concurrent language. They are

Language orientation This describes the general concepts of the language such as the computation model, the concurrency model, the intended architectures and environments, the intended problems and the overall syntax and semantics.

Synchronisation mechanism Dependening on the architecture, the processors within the multiprocessor system will require some mechanism for synchronising data access when data is to be shared.

Communication mechanism The sharing of data also requires a mechanism for transferring the data between processes.

Program modularity How the functional abstractions of a program, such as procedures and units, have been extended to deal with multiprocessor environments.

Process creation The language's ability to divide a program into concurrent processes. Also known as *program partitioning.*

Process network topology specification How a program is assigned to processors. Its flexibility to changes in the machine topology and its extendibility to different numbers of processors. How the interconnections between processors are defined and how they can be changed.

Process scheduling strategy When more processes than processors exist, the order processes are evaluated must be restricted in such a manner as to ensure

fairness and eliminate process starvation, deadlock and livelock.

Processor binding strategy The assignment of processes to processors. Also known as *process allocation.*

Process termination The manner in which processes can be terminated and how this affects related processes.

Non-determinate execution How non-determinacy is provided and controlled within a language.

Exception handling The effect of errors and other exception events on the behaviour of the overall multiprocessor system.

Ease of verification How easily a program can be proved correct including proof of termination.

Real-time support Degree of support for process control including any explicit control of partitioning, scheduling and allocation necessary to satisfy hardware or time constraints.

Ease of use The overall language should allow the user to specify a wide range of parallel examples in a natural, straight-forward manner. Changes required to introduce parallelism into existing serial algorithms should be minimised so as not to hide the underlying algorithm structure. The language should be flexible in the types of parallelism that can be invoked and not impose unnecessary restrictions on how parallelism can be defined.

The choices made for each design issue presented above determines the essence of any parallel programming language. The choices made for Scheme^{n} within each of these categories will be discussed in Chapter 2.

1.5 Concurrent Programming Languages

Having chosen a set of design specifications for the distributed language, a choice remains as to whether the language should be defined "from scratch" or whether an existing language can be extended to achieve the design goals. If an existing language can be found which has features compatible with the distributed extensions then this language can provide a useful basis for providing parallel capabilities. Extending an existing language means there will be already be people with

a clear understanding of the base language and a range of existing software. These people will be more readily inclined to use a language which for the most part is unchanged from a language they already use and understand than change to a largely unproven language. The existing attributes of a language can however conflict with the requirements of the parallel extensions. Such conflicts can limit the parallel features that can be introduced to a language without requiring major changes to the language philosophy. Care must be taken to retain the existing language style and features where applicable.

Designing a language "from scratch" has the potential of producing a language that has better performance, is easier to use and allows parallelism to be expressed more clearly than a sequential language that has been extended. When the effects and requirements of parallelism on a language have been considered during the entire design phase of the language all of the language's attributes can be chosen for good parallel performance. Also the software itself can be designed with the features of the parallel language in mind. The algorithms can therefore be chosen for their parallel performance potential.

Many existing sequential languages have been enhanced with various levels of parallel functionality, such as Concurrent Pascal, Concurrent C and HPF [2, 23]. Other languages have been designed with the central idea of providing parallel behaviour, such as Ada [42] and CSP and occam [23, 32].

Some languages have been specifically designed for various parallel architectures such as *LISP [74] for the Connection Machine. These take full advantage of the peculiarities of a particular architecture which other general-purpose languages cannot, and can produce very efficient high performance systems. The specific machine attributes can usually be emulated on other architectures, allowing programs to be ported to other machines, but these languages do not usually provide satisfactory performance in such foreign environments.

1.6 The Thesis Question

The ability to identify and take advantage of parallelism with minimal performance overhead are the key requirements of a parallel system. To accomplish this, suitable hardware and software are needed that make full use of the features of each other. They must also satisfy the requirements of the algorithm, or class of algorithms, that are the main intended application of the system.

The concurrent programming language used is the "glue" that binds the application to the multiprocessor system. The language must be flexible enough to take advantage of the available hardware resources and overcome any features lacking in the environment. It must also utilise the inherent parallelism in an algorithm without introducing unnecessary performance overheads and without forcing unnatural or unreasonable requests on the programmer. It should do all of this in a transparent manner that supports portability to other hardware environments and supports extendibility to other related algorithms.

The following thesis question is posed:

Is it possible to design a parallel language so that an implementation having the following goals might be achieved:

- fully distributed control,
- implementable on existing distributed platforms,
- performance which makes parallelism worthwhile,
- natural syntax and semantics,
- useful for general-purpose computing,
- expressive for symbolic and numeric algorithms,
- dynamic allocation and scheduling of concurrent processes but with explicit manual control also possible,
- minimal communication requirements,
- shared global variables,
- supports heterogeneous environments and a flexible processor interconnection.

The goals of the thesis question posed have been chosen with a distributed multiprocessor in mind. Distributed processing environments are more readily available through the abundance of workstation networks and can be emulated by shared memory multiprocessor systems. In addition, the author had access to a set of dedicated distributed memory processors for implementation purposes. The available multiprocessor system, consisting of a network of Inmos Transputers [35], provided a suitable development environment for detailed study of the design and performance of an implementation of the language.

The goal of fully distributed control and the avoidance of central managers in the synchronisation of the multiprocessor elements allows the resulting programming language to be applicable to massively parallel processing where centralisation can form bottlenecks as systems are scaled. Implementability and worthwhile performance allows for better evaluation of the language and justification of the design.

The syntax and semantics of the language should be chosen for conciseness, clarity and consistency. They should pose minimal restrictions on the programmer and help promote structured, human-readable code. The language should appear familiar to experienced programmers allowing many of the abstractions and constructions of sequential languages to be immediately realised. Extending an existing language to include parallel constructs will achieve this. In particular, the common use of global variables as a means of communication and synchronisation should be supported if possible. In a distributed concurrent environment shared variables have many undesirable properties but their total absence can hinder program development. Including some shared memory mechanism, for use when absolutely necessary, can increase productivity and allow easier conversion of existing sequential programs.

The language should allow concurrency to be extracted from a wide class of algorithms. Special consideration has been placed on the suitability of the language for both symbolic and numeric computation [48]. Symbolic algorithms generally contain a significant amount of *control parallelism* where the parallel activity is highly data-dependent. The operations to be applied in parallel are commonly diverse and consume irregular amounts of processing time. Their irregularity requires a programming language to be flexible in how concurrent processes are created, scheduled, evaluated and terminated.

Numeric algorithms have a more regular control structure requiring less flexibility in how parallelism is invoked but typically providing higher levels of *data parallelism* with high throughput. To maximise throughput in a known numeric algorithm usually requires manual arrangement of the processor topologies and process management to provide as close a match as possible. Developing a language that can provide features useful to both would open up fields of research where both algorithm styles are present. Without significant support for both styles the investment of time and money in a parallel implementation that only provides increased performance improvements for half of an application in unlikely to be worthwhile.

A distant goal of the author was to develop a concurrent language suitable

for use in another ongoing project at this institution. The project SENAC (Symbolic Environment for Numeric and Algebraic Computation) [11] is developing a symbolic algebra package which has significant support for numeric processing as well. Senac is currently implemented in Common Lisp and has influenced the choice of some of the many issues of a concurrent language design [12].

To increase the portability of a language and any programs written for that language, the assumptions about the capabilities and resources of the environment should the minimised. In particular, the built-in communication and synchronisation capabilities of the multiprocessor system should not be over-specified. As little reliance as possible should be placed on specialised attributes of a system including its size and configuration. Of particular note is the requirement of homogeneity of processors and their interconnection. In large distributed systems it is unlikely every processor will have the same resources and capabilities. If the differences between processors can be incorporated into the algorithm, in a flexible manner, the available resource could be utilised better.

The components of the thesis question are answered in more detail within this thesis. From the research carried out the parallel language Scheme^{n} was defined and implemented.

1.7 Scheme^{n}

Scheme [63], as a sequential language starting point for the design of a parallel extension, has many positive aspects. The language has a small clean definition that exhibits adequate sequential performance with current compiler technologies. The functional nature of Scheme allows the evaluation order of Scheme^{n} to be relaxed naturally for concurrent evaluation. The undefined evaluation order of arguments, the range of potential data parallel functions such as map and the extensive use of recursion, lend themselves readily to concurrent evaluation [1]. Reliance of other languages, such as C and Fortran, on an exact evaluation order makes the extending of these languages, to concurrent evaluation environments, more difficult. This is because data dependencies are common, forcing excessive synchronisation and sequential evaluation and more explicit concurrent control. Side effects occur in only a subset of Scheme constructs, so data dependencies are fewer or can be minimised through judicious use of side-effect-producing constructs.

Applications such as computer algebra, have a data-dependent flow of con-

trol which favours languages such as Lisp and its dialects. A parallel extension of Scheme would benefit from exploitation of the same flexibility that Scheme provides to the evaluation sequence of such sequential symbolic algorithms.

Scheme^n is designed for implementation on a distributed memory system. Communication between processors is achieved using a message passing protocol which will support medium and large grain parallelism. Processors can be connected in any topology with control of the system fully distributed among the processing elements. Each processor evaluates an independently threaded Scheme with support for implicit scheduling and allocation of tasks on any available processor. No centralised control mechanism is used, avoiding potential bottlenecks in performance. Parallelism is invoked explicitly using the future construction (first used in Multilisp [27]). The future construct has been extended to provide a synchronisation mechanism and a method for sharing data between parallel tasks.

Support for both AND and OR parallelism is central to the Scheme^n design. AND parallelism allows a task to initiate several concurrent evaluations and wait until all of the evaluations have completed before continuing. OR parallelism suspends a task until the result of one evaluation in a set of concurrent evaluations becomes available. The remaining evaluations are no longer required and can be terminated. The ability to initiate multiple parallel tasks, waiting on the success or failure of any single task or group of tasks, and terminating the remaining unneeded evaluations introduces a significant area of potential concurrency.

1.8 Parallel Lisps

Multilisp [27] was one of the first parallel lisp designs and many contemporary parallel lisp languages are based on it, including Scheme^n. Multilisp is a Scheme based language, designed for a shared address space memory model. It uses *futures* to explicitly construct a concurrent task which is automatically assigned to processors and implicitly synchronised with tasks which require its resulting value. Side effects are supported with shared objects accessible via atomic test-and-set operations, and the use of semaphores for mutual exclusion. An additional higher level construct, *pcall*, has also been implemented for fine grain concurrency between function arguments. Scheme^n is designed for medium and coarse grain concurrency and does not provide a pcall equivalent although, as with Multilisp, an equivalent facility can be implemented using futures. Unlike Scheme^n, Multilisp provides no

support for explicit process control or for AND or OR parallelism.

Qlisp [53] is another commonly modelled parallel lisp. Qlisp extends Common Lisp to support explicit parallelism in a shared address space. Concurrent processes are associated with futures that require explicit synchronisation, unlike Multilisp and Scheme^n. Processes can be explicitly spawned and killed and variables of a local environment can be evaluated in parallel using the *qlet* construct. Dynamic control of process creation, partitioning and termination allow speculative computation to be conducted while limiting excessive parallelism. Qlisp supports shared variables using a monitor-like *qlambda* construct which forces mutually exclusive access of the closure associated with the qlambda. Explicit locks are also provided to create critical sections. Scheme^n provides similar facilities to Qlisp but in a different style.

EDS Lisp [28] is a Common Lisp based parallel lisp requiring a shared address space but can be implemented on a distributed memory machine. The future construct is used to spawn parallel processes which can be explicitly killed. Critical sections and mailboxes are available for shared variable access. No support for AND or OR parallelism has been provided in EDS Lisp.

Concurrent Scheme [37] has been designed for distributed memory multiprocessors. Parallel extensions in Concurrent Scheme are based on *threads*, a concept similar to Multilisp futures. The concept of closures has been extended to the parallel environment through the use of *domains*, which enforces mutual exclusion within non-hierarchical partitions of existing closures. Each thread has singular access to a domain, which requires deadlock avoidance and may lead to the need for explicit relinquishing of the control of some domains or the generation of closures that exist in distinct domains. The management of domains to ensure correct evaluation, without unnecessarily limiting concurrency, can be difficult and counter-intuitive and this approach has been avoided in Scheme^n.

Some languages have been specifically designed for various parallel architectures. *Lisp [74] designed for the Connection Machine, uses data parallel extensions to the Common Lisp programming language to achieve parallelism. A data structure known as a *pvar* is used to provide each processor with a distinct value in a similar fashion to vector and array processors. The pvar elements are automatically assigned to available processors. Many functions, equivalent to the standard Common Lisp functions are provided that operate on all elements of the pvar in parallel. *Lisp does provide some flexibility to the size and structure of the pvars used and

the use of conditional functions does allow a limited degree of control parallelism but operating in a lock-step fashion. Operations are provided for inter-pvar access to elements and for the collection of pvar information onto a single processor. The fine grain of parallelism possible with the *Lisp design is chosen to match the shared memory architecture of the Connection Machine. There is significant communication required in the distribution of pvar information and the functions to apply to them. The data parallel paradigm of *Lisp supports a finer grained parallelism than chosen for Scheme^{n} and has not been supported.

EuLisp [8] is a dialect of Lisp that incorporates object oriented ideas and support for concurrent execution. A thread mechanism is used to initiate light-weight processes in a shared address space. Operations on shared data, requiring mutual exclusion and synchronisation between threads, must be explicitly locked. Event waiting and signal handling are available for the control of active threads. Modules have been implemented, using the available concurrency primitives, that emulate other well-known parallel abstractions. These include Multilisp's pcall and future constructs and the Linda paradigm [24]. Because of some incompatibilities between the underlying EuLisp mechanisms and the requirements of other parallel abstractions there are features of each abstraction that can not be emulated. For example, the implicit replacement of determined Multilisp or Scheme^{n} futures with their associated value is not easily realisable.

None of these parallel lisps exhibit all of the features of Scheme^{n} that were regarded as being natural and desirable for the modelling of concurrent processes in a distributed symbolic/numeric environment: namely AND and OR parallelism, explicit task killing and implicitly synchronised concurrency constructs.

1.9 Thesis Structure

When concurrency is integrated into a language implementation, additional degrees of freedom are possible in how the evaluation of an algorithm is to be conducted. The means to partition, schedule and allocate concurrent tasks to processors must be defined as does the programming models supported. Chapter 2 introduces the basic concepts of parallel processing. It categorises many of the attributes of parallel systems, highlighting the advantages and disadvantages of possible choices. From these choices the attributes of Scheme^{n} are chosen.

Chapter 3 defines the Scheme^{n} language semantics and syntax. Example

code is used to demonstrate the potential uses of these constructs. To implement the language design given in Chapter 3, a supporting parallel system is required. Chapter 4 describes such a system.

Chapter 5 introduces the environment specifics of a prototype Scheme^n implementation. The implementation has been developed on a network on Inmos Transputers [35] using the Perihelion Helios distributed operating system [54].

Chapter 6 looks at the performance of the implementation. For several simple Scheme^n programs, the performance of the Scheme^n prototype implementation, presented in Chapter 5, is examined. The effect of processor topology on performance is investigated and the performance overheads of remote invocations, communication and implicit future checking are measured. The effect of load balancing on performance is also analysed.

Chapter 7 describes an application of Scheme^n to the factorisation of large integers. Some basic algorithms are presented and evaluated for inherent parallelism. Several of these methods are then used in an OR-parallel fashion to construct a parallel factorisation package. Analysis of the runtime behaviour and performance of the factorisation package is given.

Chapter 8 concludes the thesis with a summary of the answers given to the question posed in this introduction and makes suggestions for further work that has been identified.

Chapter 2

Distributed Processing Concepts

2.1 Introduction

Before describing the selected language extensions and system design of Scheme^n in Chapters 3 and 4, this chapter will introduce background concepts of parallel processing. The concurrent language design issues, introduced in Chapter 1 will be expanded upon. The range of communication mechanisms, synchronisation primitives, and network topologies available in parallel environments is covered, and the needs and requirements of Scheme^n are stated. The strategies for program partitioning, process scheduling and process allocation in a concurrent language will be discussed. Concurrent language methods for introducing parallelism are presented and the chosen parallel primitive for Scheme^n, the future, is introduced.

2.2 Communication

Communication between processors in a multiprocessor system is the key difference between multiprocessors and uniprocessors. The manner in which processors in a multiprocessor system can exchange information is crucial in the determination of the types and level of parallelism that can be effectively supported [20].

Each communication operation between processors adds a *communication overhead* to the evaluation time of an algorithm that is not present during sequential evaluation. It is this communication overhead that can severely affect the performance of a parallel implementation. Any communication overhead must be adequately offset by an increase in performance. The hardware support for communication dictates the efficiency with which processors can communicate.

The magnitude of the overhead determines the *coupling*, or degree to which processors can effectively communicate. *Tightly-coupled* systems have a low communication overhead allowing processors to exchange information frequently. The *bandwidth* of a communication network is a measure of the amount of message traffic that can be sent in a given period. Tightly-coupled multiprocessor systems generally support a higher bandwidth of communication due to lower communication overheads. Small items of data can be exchanged frequently without causing significant degradation in processing time. Tightly-coupled systems can therefore efficiently decompose an algorithm into many small individual tasks for parallel evaluation. Such systems generally achieve tight coupling using shared memory and centralised control, for example, CMU Cm*, BBN Butterfly, Cray Research Y-MP C90, CM-2 [2, 7, 33, 62].

Shared memory systems give processors access to a common memory space allowing processors to communicate via standard memory read and write operations. They require additional synchronisation during memory accesses to arbitrate memory reads and writes from multiple processors. In particular, write accesses to a single memory location must be guaranteed to be mutually exclusive as reading a partially written value or attempting to write several values simultaneously to the same memory location can produce invalid results. Explicit synchronisation of shared memory accesses are normally required for the correct interpretation of a parallel algorithm. Synchronisation must be used to order memory accesses that are data dependent. See Section 2.4 for a discussion of shared variables.

Loosely-coupled systems incur higher time penalties for communication and must communicate less frequently to avoid performance degradation. Systems with distributed memory and an asynchronous control mechanism are typically loosely-coupled, for example, Caltech's Cosmic Cube, Intel Paragon, CM5, Ncube, KSR-1 [2, 7, 33, 62].

2.2.1 Message Passing

Each processor in a distributed memory system has exclusive read and write access to its own memory storage. Any data that is required by other processors must be explicitly copied through *message passing* [62]. Message passing involves sending specified information via the communication mechanisms available. It requires explicit programming, additional to standard sequential code, to invoke the communication. The data involved is normally consigned to a contiguous block

of memory which can be transferred to an equivalently sized block of memory of the destination processor in one operation. The information must be converted into a format suitable for transmission. In particular, the memory address of the destination memory block is not usually guaranteed to be identical to the source memory address so that any absolute memory addressing contained within the copied data will be invalid unless it is translated. The formatting and transmission times tend to be significantly larger than equivalent local data access (milliseconds rather than microseconds). This reduces the bandwidth and lowers the effective communication level.

Message passing can be synchronous or asynchronous in nature. *Synchronous* communication requires the source and destination processes cooperate in the exchange of data. The source process cannot proceed with a data transfer until the destination process is ready to receive the incoming data. CSP and occam [13, 32] use synchronous communication.

Synchronous communication imposes an implicit synchronisation between the sending and receiving processes requiring a one-to-one association between read and write operations. This can be useful for proving a program's correctness, and in situations where the communication channels may not provide error-free communication, synchronous communication can also help pinpoint the places where data has been sent but not received. A disadvantage of synchronous behaviour is the need to delay processes whenever the source or destination are not ready to communicate. Synchronisation among processes can be over-specified reducing the performance of the system.

Asynchronous message passing does not require the destination process to be ready to receive the data and as such there is no implicit synchronisation delay. A receiving process must of course still wait if the expected message has not yet been sent. Communication is buffered to allow the source process to continue. This requires additional hardware or software support to manage the communication buffers [62].

Given hardware that communicates in a synchronous manner, it is possible to emulate asynchronous behaviour, and vice versa, using communication protocols whose behaviour is asynchronous or synchronous respectively. Asynchronous behaviour can be emulated using additional memory buffers and multitasking lightweight processes, and synchronous behaviour is possible through explicit waiting for additional acknowledgement messages.

Scheme^n approach: The Scheme^n language has been designed for distributed memory systems although processor-to-processor communications are not directly supported. At the language level, communication is supported in a functional style through procedure arguments and results and through higher level data constructs such as streams (see Chapter 3). Some shared memory characteristics are also provided in Scheme^n (see Chapter 4) to communicate globally accessible data.

Direct processor-to-processor communication is used internally to transfer data, perform system synchronisation and control task allocation. Scheme^n requires asynchronous behaviour to receive an unknown number of messages from a variety of sources while continuing to process its own tasks (see Chapter 5).

2.2.2 Message Passing Environments

Many implementations of message passing interfaces have been created to provide the communication mechanisms necessary on distributed memory systems. Some of the massively parallel processing systems (MPP) based on message passing include Caltech's CROS, IBM's EUI, Intel's NX, Meiko's CS, nCUBE's PSE and Thinking Machine's CMMD [47]. Similar services for synchronisation and process communication exist in each of these systems but differences occur in how these features are made available. The features of each system have been carefully chosen to match the capabilities of the vendor's parallel hardware. Converting software from one platform to another is usually possible, as equivalent capabilities are present in each environment, but the manner in which the communication services are expressed can require considerable effort to translate.

To increase the portability of parallel software, a more architecture independent environment is needed. The portable environment should provide the same functionality as the hardware-dependent message passing systems but do so without confining an implementation to any specific architectural requirements except in the broadest terms. For practical purposes, the environment defined must also be implementable on a sufficiently wide range of existing computing platforms. A portable message passing environment can assume only the simplest message passing mechanisms are available, such as send and receive, and must construct a richer set of services from these.

Some of the available portability platforms for message passing environments include RPROC, EXPRESS, Linda, P4, PARMACS, PVM, and Zipcode. RPROC, PVM, PARMACS, EXPRESS and Zipcode are all based on the send-

receive communication model with added support for global communication. P4 uses a mixture of point-to-point and shared memory communication. PVM, P4, PARMACS and Express also include additional support for process management and load balancing. Linda [24] uses message passing indirectly to create an associative virtual shared memory system. The interested reader is directed to [24, 52, 73] for further information on each system. The ability to give different machines a unified interface also means that portability platforms are capable of supporting heterogeneous computing on a network of differing computer architectures and operating systems.

The existence of many incompatible portability platforms has seen the proposal of a standard message passing interface for MIMD distributed memory systems. The Message Passing Interface (MPI) standard [75] draws ideas and concepts from many existing portable and dedicated message passing environments and consolidates them into a general purpose interface for message passing. The proposed standard does not define a complete parallel processing environment and issues such as process management, load balancing, remote memory transfers, threads, parallel I/O and virtual shared memory are not considered.

The MPI standard provides point-to-point communication in blocking and non-blocking modes and collective communication for broadcasting and other globally coordinated operations. The data types used in messages can be constructed from a set of primitive data types allowing a wide range of information to be exchanged between processes. The scope of messages can be defined to provide better control of message destinations. The concepts of process groups and messages context are used for this purpose. Process groups allow the collective communication operations to be applied to subsets of the complete set of all processes. They also allow groups of processes to evaluate distinct parallel tasks. Message contexts associate tags to messages to ensure messages are received by the relevant subroutines of the destination process. It is especially useful when considering shared libraries that use message passing.

Scheme^{n} approach: While the current implementation of Scheme^{n} uses machine-specific message passing routines, the additional abstraction of the Scheme^{n} concurrency primitives should allow a user-transparent translation to other message passing environments. See Chapter 5 for a discussion of the hardware and software requirements of Scheme^{n}.

2.3 Synchronisation

When multiple processors are interacting in the evaluation of some computation there is a need for these processors to synchronise their activity. For asynchronous computers, synchronisation is an explicit operation. Activity on each processor is usually allowed to proceed independently of the activity of others. Synchronisation may be required, for example, in order to communicate necessary information, to provide mutually exclusive access to some shared resource, or to control the relative priorities of each processor's actions.

Synchronisation can be used to ensure fairness of activity among processors and avoid potential starvation of processes from resources. Inappropriate synchronisation can also cause deadlock or livelock of a program. Over-specification of synchronisation can reduce the available parallel activity by delaying a processor from continuing through waiting for unnecessary synchronisation with other busy processors. Under-specification can allow invalid operations, or sequences of operations, to be performed, resulting in incorrect evaluation.

Where synchronous message passing is used for direct communication, for example occam [36], pairwise synchronisation of the communicating processors is implicitly imposed. The send and receive operations automatically synchronise the evaluation of the communicating processes. For asynchronous message passing, synchronisation can be introduced using a handshaking protocol involving bi-directional communication. Because of the independence of processors in a message passing system, synchronisation is only required during message communication.

For systems using shared variables, the interaction between processors is potentially greater, increasing the synchronisation requirements. The indirect communication possible through the access of shared variables sees the need for more explicit mechanisms for synchronisation to avoid unexpected combinations of operations. Mechanisms such as busy waiting, semaphores, monitors, conditional critical regions and path expressions [3, 31, 42] provide different methods of mutually exclusive access to shared variables, or other resources.

Scheme^n approach: Scheme^n uses a combined message passing and shared variable synchronisation method. Interaction between Scheme^n processes is through procedure evaluation that can be initiated on different processors (see Chapter 3). This functional style requires less synchronisation than that required by shared variables but because the communication is indirect, through the future construct (see

$$\begin{array}{c} \{v{=}1\} \\ \text{(set! v (+ v 2))} \parallel \text{(set! v (+ v 1))} \\ \{v{=}?\} \end{array}$$

Figure 2.1: Serializability

Section 2.8), more synchronisation than message passing is needed.

Scheme^{n} also provides shared variables and, for their proper use, synchronisation mechanisms of busy waiting, semaphores, or monitors can be used (see Chapter 3).

2.4 Shared Variables

Although the use of shared variables, when used in a distributed environment, should be avoided where possible, by directly passing necessary values as procedure arguments or modifying the algorithm to remove or reduce the number of accesses, there may still be instances that require shared variables.

Shared memory can be emulated in a distributed memory system and vice versa. The Linda paradigm [24] is an example of a virtual shared memory mechanism that requires only distributed memory hardware to provide shared memory access. Emulating distributed memory on shared memory hardware can also be achieved using data structures such as mailboxes [33] to simulate the sending and delivery of information through a named address.

2.4.1 Serializability

When variables in a distributed environment are to be accessed by more than one process in an indeterminate order and those accesses involve modification to the variable's value, the *shared variable* must be accessed in such a way as to provide consistency. Consistency of a variable requires all processes use the same value for the variable and that the result of multiple modifications performed concurrently are equivalent to some sequential ordering of the component modifications. The implied sequential ordering of variable modification, known as *serializability*, is important in proving the correctness of a concurrent program [51].

An illustrative example of serializability can be seen in Figure 2.1. If these two instructions were performed concurrently, and both instructions were to read

$\{a{=}1 \quad b{=}5\}$
(set! a (+ b 1)) || (set! b (+ a 1))
$\{a{=}? \quad b{=}?\}$

Figure 2.2: Serializability between multiple variables

1 as the current value of v, to which 1 or 2 is added, then the value of v after both instructions complete could be either 2 or 3 depending on which instruction performed its assignment last. Neither of these two results for v would be obtained if these same two instructions were performed sequentially. Independent of the sequential evaluation order, the serial result for v is 4. If these two instructions can evaluate in such a manner as to produce a result of 2 or 3 as indicated, then the correctness of the program would depend on exactly how these two instructions are evaluated and its serializability is compromised.

To ensure serializability in the example of Figure 2.1, there can be no overlap in the instruction evaluations between the reading of the shared variable and the subsequent reassignment. Provided one instruction reads the value of v assigned by the other instruction then serializability is attained.

Similar serializability behaviour can occur between several variables. Figure 2.2 illustrates the case for two variables a and b. Depending on the sequential order chosen for the evaluation of the two instructions of Figure 2.2, then two valid sequential results for a and b are a=6,b=7 or a=3,b=2. Provided either of these two combinations is attained in a concurrent evaluation the behaviour of the program is more easily modelled and understood. However, if each instruction is able to read the value of a variable before the other instruction has reassigned it then a value of a=6,b=2 is possible in a concurrent evaluation.

2.4.2 Emulating Shared Memory

When shared values are fixed and will never be modified the data objects can be distributed among the processes by broadcasting a copy of the value. Each processor can then access its own local copy of the data without any need to interact with other processors. As the data object is static and does not change its value the initial broadcast of the value is the only overhead. If the shared value is used infrequently or by few processes broadcasting can be inefficient. In these situations the shared value can be distributed to processes individually as the process requires

it. Once the process has acquired the shared value all further accesses to that value can be performed locally.

Maintaining distributed copies of mutable data is much more difficult. When the value of an object is to be modified all copies of the data existing in the system must be correctly adjusted to the new value. Problems with synchronisation arise as processors may be using old copies of the shared data object during the period when the value of the object has been modified but the new value has yet to be distributed. To avoid the possibility of several values for an object being simultaneously used in a system, synchronisation amongst all processors is required. This will entail broadcasting a message to all processors indicating a change to the data value is required and waiting until each process acknowledges that it is safe to make the modification. The processor wishing to modify the shared data can then make the required changes and broadcast the new value to all processes, allowing them to freely access the object again. The problem of synchronisation is further complicated when several processors can concurrently attempt to modify shared data values. In this situation, some mediation is required between processors to determine which process has priority. Priorities can be predetermined or they can be decided by some independent process.

The alternative to maintaining copies of common data in a distributed environment is to assign the data to a single processor but allow other processors free access. Single copies of data requires any processor wanting access to communicate with the local processor. The problem of modification is simplified when only one copy exists as the local processor can mediate between conflicting operations on the data object. If the value is accessed frequently the communication delays incurred for each access can become significant and adversely affect performance. Care is still required to consistently modify a set of shared data, especially if this data is assigned to different processors. Deadlock can occur if processes are mutually waiting on access permission to shared data while holding access on data required by the other process.

Scheme^n approach: All current lexical variable values of a remote procedure are taken from the calling environment. These are copied to the remote task as required. By default, the variables are assumed to be read-only. Modifications by the source process after the variables have been copied will not be seen by the remote task and changes by the remote task will not be seen outside the remote environment (see

Chapter 4).

Mutable shared variables can be created in Scheme^{n} (see Chapter 3). All remote access to these variables is referred back to the single copy of the variable on the source process. Additional semaphore synchronisation primitives are provided to correctly control the multiple asynchronous accesses that can occur with such variables (see Chapter 3).

2.5 Topology

In a multiprocessing system with distributed memory and using message passing for communication, it is beneficial to have a well-defined arrangement of dedicated communication channels to extract the best performance. Using a dedicated network for communication can increase the bandwidth possible between processors allowing a finer task granularity than an unstructured communication medium, such as a local area network, without the need for shared memory.

The processor arrangement, or *topology*, chosen depends on both the hardware properties and on the classes of algorithms intended for the system [2]. The degree to which processors are connected can vary from minimal connectivity, where as few as possible physical connections are used to provide a limited communication ability, to a fully connected system with a full complement of connections. Within this range commonly used topologies such as pipelines, rings, two and three dimensional grids, hypercubes, shuffle-and-exchange networks, and n-ary trees, provide various levels of connectivity [2, 33, 62]. These topologies provide an extendible, regular interconnection of processors. Combinations of these and other topologies can also be used in a hierarchical fashion.

Topologies are chosen to utilise the connection abilities of a set of processors. In many cases the processors in a system are identical and a regular topology is chosen to reflect this and to give each processor a balanced workload with equal communication privileges. It is usually a design goal to have the potential to increase the number of processors in a multiprocessor system to achieve a proportionate increase in the system performance. To aid future increases in processor numbers, a regular topology which is easily extendible is advantageous. If the connection requirements of each processor are not directly related to the number of processors in the multiprocessor system then extending the system will not require changes to existing processor interconnection capabilities.

Full connectivity, while computationally desirable, does not usually provide a cost effective means of processor interconnection as it is unlikely all connections will be fully utilised and the sheer number of communication paths that must be provided would require expensive technology. Providing connections to this degree can also meet serious physical constraints such as power supply demands and limitations to the number of pins on a processor integrated circuit. For these reasons, topologies with fewer connections per processor are the norm.

Choosing a topology is an important multiprocessor design decision as the hardware topology of most current multiprocessor systems, once chosen, is fixed. Additional hardware can be added to the system to allow the communication links to be controlled by software and thus allow the topology to change dynamically but this can be expensive and can contribute to hardware reliability problems.

Several metrics can be used to measure the expected performance of each topology including:

- average number of connections per processor,
- average distance between processors and
- maximum distance between processors.

where distance refers to the number of direct processor-to-processor links necessary to communicate between a given pair of processors. But as the performance of a topology is highly dependent on the logical communication requirements of the algorithms involved, these metrics can only serve as a guide.

Where the performance of one algorithm type is of prime importance and the hardware can be dedicated to the evaluation of this one type, the processor interconnections, and even the processors themselves, can be designed to match the specific attributes of the algorithm type. Modifying the hardware to match a particular algorithm has only limited appeal in a general computing environment and especially in a multi-user environment where each individual user will have his or her own preferences for the communication topology and these preferences are likely to conflict.

Scheme^n approach: Scheme^n does not require a homogeneous topology. A logical topology is used for all communication. This topology, fixed when the Scheme^n system is started, is assumed to reflect the underlying hardware topology. The number of processors connected to any processor is unlimited, allowing both minimal

and fully connected topologies to be supported. Chapter 6 analyses the effect of different logical topologies on the performance of Scheme^n.

2.6 Partitioning, Scheduling and Allocation

To begin concurrent evaluation, several processes must be started on multiple processors. The number of processors used for the evaluation can be statically defined by the program, automatically allocated by the operating system based on processor availability, or dynamically controlled during program execution. The number of processes that can be defined need not be limited to the number of processors available. Provided there is operating system support for multitasking, several processes may be created on each processor. Depending on the algorithm, the exact number of processes may or may not be important. An algorithm may either require a predefined number of processes, which is independent of the data being evaluated, or the number may be statically determined from input data values or sizes, or process numbers may dynamically vary during the program evaluation.

To support dynamic process requirements either sufficient processors must be allocated at initialisation to satisfy all future needs, or the operating system and programming language must support the addition of processes to an evaluating program. This may require reallocation of existing processes, the runtime determination of processor numbers and communication paths, or modifying access priorities to shared memory.

Given a program to solve some computational problem, a parallel solution requires the following additional operations in order to extract concurrency:

Program Partitioning The program must be divided into components that can be evaluated in parallel,

Task Scheduling The moment when each partitioned component should begin evaluation must be chosen, and

Process Allocation Individual tasks need to be assigned to processors for evaluation.

To take the best advantage of available parallelism requires a close mapping between computer resources and the parallel components of an algorithm. The number of parallel components and their processing and communication requirements

need to coincide closely with the hardware capabilities. There will be a performance overhead imposed by the system in processing the parallel tasks as each task is scheduled and allocated to a processor for evaluation.

The regularity of an algorithm, in terms of task size and communication requirements, determines the flexibility necessary during partitioning, scheduling and allocation. An algorithm with a well-defined structure that matches the hardware topology can be statically mapped at compile-time. For example, data parallel algorithms such as vector addition, where the parallel tasks of adding a single vector element have identical evaluation times and communication needs, can be trivially partitioned, scheduled and allocated for best performance on a vector processor with sufficient processors to assign each task distinctly. Algorithms with ill-defined evaluation times, irregular communication patterns or those with requirements that do not match the available hardware, will require less trivial solutions to task creation, initiation and assignment. A more dynamic approach to these operations may be needed in order to effectively parallelise the program at run-time.

Each dynamic method may also be performed using a centralised or distributed approach. A *centralised* approach assigns the task of partitioning, scheduling or allocating to a single processor. This processor has sole charge of the relevant action and is supplied with all pertinent information to make the decision. Because of its unique role, the processor can become a bottleneck to computation when excessive requests are made of it. *Distributed* schemes do not rely on a single processor and instead divide the responsibility amongst all processors. In doing so, such schemes can lose some control over the actions applied as each processor acts semi-independently, using potentially incomplete and out-of-date information. Distributed approaches have the advantage of locality and do not need to synchronise or communicate with possibly distant processors.

Scheme^n approach: Scheme^n uses a combination of static and dynamic methods to perform the partitioning, scheduling and allocation operations. Scheme^n supports only a static definition of process numbers, processor allocation and topologies. Any number of concurrent tasks are supported through forked *light-weight processes* (LWP) which multitask using shared code and data. There is also a goal to achieve distributed operation in each case to avoid dependence on a single processor for scalability and reliability reasons.

2.6.1 Program Partitioning

In order to evaluate a program in a parallel manner, the program must be divided, or *partitioned*, into groups of its basic operations. Each group, known as a *task* or *job*, can then be evaluated in parallel. Tasks can consist of a single basic operation or they can be comprised of a sequence of operations or other sub-tasks that can also be evaluated in parallel. Because of data dependencies present between the basic operations in an algorithm, it is not usually possible to evaluate every operation in parallel even if there were processing units available to do this. The partitioning goal is to divide the algorithm into tasks with few data dependencies, to avoid delays and make good use of the available hardware resources.

The degree to which a program is divided into tasks is termed its *granularity*. If the number of tasks is high and the amount of computation involved in each is small then the program has *fine* granularity. Whereas, if the evaluation of each task requires significant computing resources the granularity is *coarse*. If a parallel system has many processing units, especially if the processing power of each unit is limited and the overhead of initiating tasks is small, a finer granularity is suggested.

Scheme^n approach: The tasks of Scheme^n are manually defined using the procedure declarations of the program and the `remote` procedure call (see Chapter 3). Although the scope of an individual partition is fixed by the static procedure definitions, the procedures which form the parallel partitions are dynamically controlled by the occurrences of the `remote` procedure.

2.6.2 Task Scheduling

Given a set of partitioned tasks it is then necessary to decide when each task should begin its evaluation. The decision is based on the the data dependencies present between the tasks and the expected duration of each task. The earliest a task can be scheduled is determined by the availability of data it requires from other tasks. A good schedule of tasks will minimise the number and duration of delays caused by tasks waiting for data that is currently unavailable.

For best performance all tasks that can execute should be started as soon as possible to keep all processors busy. If there are more active tasks than processors then tasks whose results would delay further calculation the most should be given priority. These tasks form a *critical path* of evaluation within which any delays

would lengthen the execution time of the algorithm. Tasks not on a critical path have a certain degree of flexibility in their evaluation time before the determination of their results becomes critical.

In any schedule the lack of exact task evaluation times usually means scheduling can not be performed optimally as critical paths and their associated tasks cannot be found. In the absence of evaluation times, the scheduling decision can be delayed until run-time by prioritising the need to evaluate particular tasks. The ability to perform speculative computation, where some computations may be unnecessary, introduces additional concerns for the control of the computation. Manually assigning priorities can help choose which computations should be given the most processing resources [50].

Scheme^n approach: The scheduling of tasks in Scheme^n is implicitly controlled through the manually declared partitioning induced by the `remote` procedure. Where possible, tasks are scheduled for evaluation as soon as they are declared. When no processor is available to evaluate the task, the task is implicitly delayed until such time as resources become free or until the result of the task is required (see Section 2.8). No priority is assigned to tasks and competing tasks are scheduled in an undefined order. Speculative computation is controlled through explicit termination.

2.6.3 Task Allocation

Having decided on the program partitioning and the task schedule, each task must be assigned to a specific processor for evaluation. The important criteria in task allocation is to try and maximise the utilisation of processors. *Load balancing* is the mechanism used to achieve an even processor utilisation [14, 19, 21, 26, 29, 44, 45, 68]. The *workload* or *load* on a processor relates to the number of tasks currently being evaluated by that processor plus the computation throughput achievable. Heavily-loaded processors require more time to compute an evaluation than a lightly-loaded processor due to the other processing commitments the processor currently has.

Static allocation techniques choose the processor for a given task at compile time. This can lead to a very uneven balance amongst processors when some tasks complete early, leaving the processor idle, while other pre-assigned tasks wait for evaluation by busy processors. Dynamic task allocation techniques delay the allocation decision until the task is to be assigned, using the current load informa-

tion of processors to determine the best allocation. Some allocation methods go even further to achieve an even load balance by reassigning running tasks from fully loaded processors to processors with lighter loads. The benefits of such *task migration* must be balanced against the overhead and additional complexity in process management.

A secondary consideration in task allocation is to minimise the communication overhead by assigning communicating tasks to processors 'close' to each other. By keeping the communication distances short the overhead of task evaluation is reduced by minimising communication traffic and hence avoiding delays from possible communication contention.

Scheme^{n} approach: Scheme^{n} uses a distributed dynamic and automatic method of scheduling. No migration of active tasks is attempted and the allocation can be manually overridden. See Chapter 4 for details of the Scheme^{n} load balancing mechanism.

2.7 Methods of Introducing Parallelism

Within a system of dynamic processes, a means to construct communicating tasks must be provided [2]. One method involves specifying the set of concurrent tasks to be evaluated. The statement

parbegin $P_1; P_2; \ldots; P_n$ **parend**,

can be used to initiate the concurrent evaluation of the independent tasks P_1, $P_2, \ldots, P_n$. The task executing this command will continue only when all component tasks have completed. The component tasks can be as simple as individual programming statements or as complex as complete self-contained programs, depending on the language and implementation. Such constructs are limited by the static grouping of the concurrent tasks and the single entry and exit access to the control structure. The languages occam [36] and Concurrent Pascal [10] provide this method of process creation.

A dynamic number of similar parallel tasks can be initiated with the `doall` construct, or equivalent. This construct, similar to the sequential loop iterator, creates multiple concurrent tasks which differ in the value of the `doall` variable.

For example, a statement such as

$$\texttt{doall i} = \texttt{1,N } P(i) \texttt{ end}$$

creates **N** tasks P_1, P_2, ..., P_N, where each task P_i is passed the value i. Data parallel languages, such as HPF [30], make extensive use of variations of such parallel replication.

A more flexible notation is *fork-join* commonly used by multitasking operating systems such as Unix [20]. The *fork* operator duplicates the executing task with the only distinction between tasks being the value returned by the fork operator. The *join* operator synchronises the original task with the completion of the forked task by checking the value of a shared variable. Multiple tasks can be forked and then joined in an arbitrary order. Although flexible, the lack of structure of the fork/join mechanism can lead to programming difficulties.

More structured parallel initiation methods rely on the program structure to simplify their use. Constructing tasks from declared program objects, similar to procedures, provides a named environment of processes that can be readily identified. These components can then be initiated, by name, as required, allowing a dynamic set of various tasks to be evaluated in parallel. Ada [66] uses this method of process initialisation.

Concurrent tasks can also be assigned identification as they are initiated, rather than during program definition, by identifying the tasks with variable values. The variables can then be used to interact with the concurrent tasks, for communication or synchronisation. The control of tasks initiated in this manner is more structured than fork-join, as tasks are associated with variables and can be manipulated directly, while still retaining the flexibility of dynamic initiation. Scheme$^{\mathrm{n}}$ uses this method to introduce concurrency through the **remote** procedure and the *future* construct.

Scheme$^{\mathrm{n}}$ approach: Futures form the core task initialisation, communication and synchronisation mechanisms of Scheme$^{\mathrm{n}}$. The future data object encapsulates the dynamic set of tasks that are carried out concurrently. Section 2.8 describes the properties of futures and how they can be used for parallelism.

2.8 Futures

Futures are an extension of the lazy evaluation model of functional programming [1] and used in earlier parallel lisp implementations such as Multilisp [27] and Qlisp [53]. Rather than delaying evaluation until the result of the evaluation is required, as in lazy evaluation (cf. the `delay` procedure of Scheme), futures allow tasks to be actively evaluated in advance of their being required. Only tasks which depend on the result of an initiated future task are required to suspend their evaluation when the future result is needed. Other unrelated tasks can continue processing independently of the evaluation state of the future.

2.8.1 Future Concept

A *future* is a data object which represents the result of an evaluation. The future is specifically designed to act as a temporary placeholder for the result that is actively being evaluated. By associating a future with a concurrently evaluating task, an undetermined future can be used to produce concurrency. When a concurrent task is initiated, an *undetermined future* is immediately returned as a temporary result, allowing the local task to continue evaluating other operations which are independent of the remote task evaluation.

An *undetermined future* has no currently defined value. It is associated with an evaluation by a processor-unique identification number which, together with the processor id, forms a system-unique identifier. The evaluation is expected to return a value which, when it becomes available, will be assigned to the associated future forming a *determined future.*

By referencing the future, the result of the associate remote task can be accessed. If the remote task has already completed when the future is accessed, the determined future can immediately provide the result. Accessing a future which is currently undetermined will require the local task to be suspended until the remote task completes and the future is determined.

The method of synchronisation between an uncompleted task and another depends on whether the futures implemented are implicit or explicit. *Explicit futures* require the user to explicitly request the value associated with a future. Such requesting is termed *touching* and forces the local task to delay further processing until the value of the future has been supplied. The concurrent lisp languages EuLisp [8] and TLisp [40] use explicit futures.

The use of *implicit futures* to represent values is transparent to the user. Both undetermined and determined futures are accessed in an equivalent manner to any other data object and can be used in any place that the associated evaluation result could be used. The implicit resolution of futures hides the evaluation mechanism of a value. References to a value should be unconcerned with when, where and how the value was derived. Such knowledge violates the important information hiding principle of computer programming [46] and can force a closer binding between otherwise distinct sections of a program. Chapter 4 explains the additional language requirements of implicit futures.

In addition to allowing the initiating task to perform operations which are independent of the remote evaluation, the immediately returned undetermined returned future can be used by the initiating task, in lieu of the actual value it represents, for certain operations where the operation does not depend on the value of its arguments. Futures can be used without their actual value being required in the following situations:

- In argument passing.
- In variable assignment.
- As arguments of non-strict procedures.

Futures need not be determined when passed as arguments to a procedure. The reference that the future provides is sufficient until an operation dependent on the value is applied. Some Scheme syntax, such as `if`, `cond` and `and` are not strictly procedures and arguments of these constructs must be evaluated in the appropriate order [63]. For variable assignment, a future can be assigned to a variable in place of the actual value as it can be replaced when the true value of the variable is required.

There are also non-strict procedures in Scheme which do not require exact values as arguments to perform their operations so placeholders can be substituted at a later time. These operations include the basic `cons` procedure, the `append` procedure and the `list` procedure. Each of these operations takes data objects as components to construct lists. The values of the components of the lists are irrelevant to the list-making operations. For the `append` operation only the last argument can be represented by an undetermined future as all other arguments, which must be lists, are copied to create the resulting list with the last pair node of each list modified to join the lists together. To find and modify the last element of a future value requires its true value. The `append` operation therefore allows only

the last list to remain as a future and implicitly touches all other arguments during processing.

2.8.2 Future Types

Scheme^{n} uses the following types of futures: Undetermined, Determined, Terminated, Terminated-with-no-value, Rejected, Lazy, Shared, Error. As already discussed, the undetermined and determined future types are necessary to all future implementations. The terminated and terminated-with-no-value futures have been added to Scheme^{n} to implement explicit termination of remote procedures. The rejected future is used in controlling the processor overloading that may occur during dynamic load balancing. Lazy futures have been introduced to allow more efficient communication of large or seldomly used data and to give read-only global access to lazy variables. Shared futures provide read-write global access to shared variables, emulating a shared memory environment. Error futures provide some distributed error handling features. Chapter 4 describes the detailed implementation needs of each future type.

Chapter 3

Scheme^{n} - A Distributed Language Design

3.1 Introduction

The previous chapters have introduced the field of parallel processing and the design issues relevant to a concurrent language. This chapter describes the Scheme^{n} language syntax and semantics. The concurrent extensions to the Scheme language are defined. For each added procedure, its syntax and semantics are given. Simple programming examples are also given to help illustrate how these constructs can be used.

3.2 Scheme^{n} Parallel Constructs

Scheme^{n} is an extension of Scheme as defined in the third revision of the report on Scheme [63]. A number of additional procedures have been added to those already present in Scheme to provide the necessary parallel processing capabilities. This section describes these additional procedures to the extent necessary for their use in a Scheme^{n} program. Examples of their use are given in the following section.

Since the initiation of this research, an IEEE and ANSI standard for Scheme has been defined [34]. No attempt has been made to ensure the Scheme^{n} constructs are compatible with this newer standard or the fourth revision of the Scheme report [64], although no problems are foreseen.

3.2.1 remote

The procedure **remote** is the central means of introducing parallelism into a Scheme program. The **remote** syntax is similar to the built-in Scheme procedure **apply** and has, as arguments, a procedure and any arguments of that procedure. The syntax of **remote** is

(**remote** (*predicate destination*) *procedure arguments*)

where *destination* and *arguments* are optional. The alternate syntax

(**remote** *predicate procedure arguments*)

can be used when the *destination* argument is not provided and *predicate* does not evaluate to a list.

When **remote** is evaluated, the **remote** argument *predicate* is first checked to determine if *procedure* should be evaluated as a remote task. If *predicate* is false then the remote procedure call is equivalent to **apply** with *procedure* applied to *arguments* by the local task, and the result returned as the result of the **remote** call.

If *predicate* evaluates to true then *procedure* will be evaluated as a remote task (see Chapter 5). The remote call will immediately return an undetermined future as its result (see Chapter 4). This allows the local task to continue before the evaluation of *procedure* has completed. The undetermined future represents the result of the *procedure* evaluation and reference to this object will return the procedure result.

The evaluation of the remote task is usually automatically and dynamically allocated to some available process of the Scheme^n system. Allocation depends on current processor activity. Other processing activity can cause the remote task to be evaluated by the same process which invoked it, or evaluation may be delayed until processor activity decreases (see Chapter 4).

Task allocation can be explicitly controlled by supplying the optional argument *destination*. The *destination* argument specifies the process identifier, assigned at system initialisation, of the process on which the task will be evaluated if *predicate* is true.

3.2.2 touch

The syntax of **touch** is

(**touch** *expression*)

where *expression* is any valid Scheme^n expression.

If *expression* evaluates to an undetermined future then **touch** waits until that future has been determined before returning the future's value as a result (see Chapter 4). If *expression* is not an undetermined future **touch** does nothing, returning the evaluated *expression*.

The **touch** procedure is seldom required as implicit touching is performed on futures when the determined value is required (see Chapter 4).

3.2.3 future?

The syntax of **future?** is

(**future?** *expression*)

where *expression* is any valid Scheme^n expression.

If the *expression*, when evaluated, is currently an undetermined future then **future?** returns **#T** otherwise **future?** returns **#F**. Unlike other procedures, **future?** does not require the value of the future to be determined and will always return immediately. However, during the evaluation of *expression* any values of undetermined futures needed to complete the evaluation will be touched.

The **future?** procedure allows an algorithm to check the status of a remote evaluation without waiting for the evaluation to complete. Busy waiting on multiple futures can therefore be implemented allowing a program to use the execution status of tasks to control the program's behaviour. When used in conjunction with the **kill** procedure (see Section 3.2.12), remote evaluations that are no longer required can be terminated at any time after their initiation. Synchronisation of multiple remote tasks can also be achieved using the **parallel** and **wait** constructs (see Section 3.2.10 and Section 3.2.11).

3.2.4 lazy

The syntax of **lazy** is

(**lazy** *expression*)

where *expression* is any valid Scheme^n expression. The result of the **lazy** procedure is semantically equivalent to the value of the expression. The **lazy** procedure is

used, in conjunction with **remote**, to delay the sending of *expression* to a remote task. By specifying an argument of the **remote** procedure as a lazy expression, the *expression* value is not transmitted immediately to the remote task. When the value of the associated argument variable is required by the remote task, the *expression* value will be supplied (see Chapter 4).

For example,

```
(remote #T cons 1 (lazy long-list))
```

which is semantically equivalent to `(cons 1 long-list)`, performs the concatenation of 1 to the value of `long-list` as a remote task. Due to the **lazy** construct, the `long-list` value is not immediately transferred to the remote task, and since **cons** is a non-strict procedure, the actual value is not required remotely. The resulting pair `(1 . lazy-long-list)` can be returned as the result of the remote without ever sending `long-list` to the remote task. When the resulting list is subsequently accessed, the lazy expression variable `lazy-long-list` will be implicitly replaced with the actual value.

Provided the lazy expression is not modified between the time of the **remote** procedure call and the subsequent transmission of the lazy expression value, the **lazy** procedure has no semantic effect and is only used to improve the efficiency of passing large data structures as part of **remote** procedure calls. Transmission of the lazy expression can be initiated explicitly, by either source or destination tasks, using the **force** procedure (see Section 3.2.5).

When *expression* is a procedure and the resulting lazy future is used as in a procedure call by a remote task, the lazy future is treated differently. In this instance, the lazy procedure call is sent to the source process, together with any arguments, for evaluation. The remote task is forced to wait until the evaluation is completed. The lazy procedure call is equivalent to a remote procedure call, where the destination is implicitly determined by the source of the lazy procedure, that is immediately touched.

Lazy procedure calls can be used to avoid sending large quantities of information to remote tasks where the information is only used by a single procedure call, and that procedure is computationally trivial. For example, a procedure that searches a large database to find a single requested entry can avoid sending the entire database between processes by specifying the procedure as a lazy future and therefore performing the search at the database site. Another situation where lazy procedures may be useful, occurs when significant use of closure variables is made.

To avoid sending the contents of a procedure's closure, the procedure can be evaluated on the process containing the closure and only the result returned.

3.2.5 `force`

The syntax of **`force`** is

(**`force`** *expression*)

where *expression* is any valid Scheme^{n} expression. **`force`** is used in conjunction with the **`lazy`** procedure to explicitly initiate transfer of the delayed communication of the lazy expression. The result of **`force`** is undefined. If *expression* is not a lazy expression then **`force`** has no effect.

3.2.6 `make-semaphore`

The syntax of **`make-semaphore`** is

(**`make-semaphore`** *count*)

where *count* is an integer. This procedure constructs a semaphore for use as a synchronisation primitive. The argument *count* sets the initial value of the semaphore. Positive values will allow *count* calls of **`request`** (see Section 3.2.7) to be immediately satisfied. Negative values will require *count* calls of **`signal`** (see Section 3.2.8) before a **`request`** will proceed.

3.2.7 `request`

The syntax of **`request`** is

(**`request`** *semaphore*)

where *semaphore* is a semaphore created by **`make-semaphore`**. **`request`** decrements the associated semaphore count and if the count is negative will suspend the task. The result of **`request`** is undefined.

(**`request`** *var*) can also be used to request exclusive access to the shared variable *var* (see Chapter 4).

3.2.8 `signal`

The syntax of **`signal`** is

(**signal** *semaphore*)

where *semaphore* is a semaphore created by **make-semaphore**. **signal** increments the associated semaphore count and will restart any tasks waiting on the semaphore if the count is positive. The result of **signal** is undefined.

(**signal** *var*) can also be used to release exclusive access to the shared variable *var* (see Chapter 4).

3.2.9 make-shared

The syntax of **make-shared** is

(**make-shared** *var*)

where *var* is a variable identifier. The result of **make-shared** is undefined. The value of *var* is unchanged but **make-shared** modifies the behaviour of the variable *var* to act as a shared variable when used in remote tasks (see Chapter 4). After the **make-shared** declaration, any modifications to *var* by any task will be observed in all other tasks. This differs from the standard remote task modification where changes are not reflected outside a remote task's environment.

For example, consider the statements

```
(set! v 1)                                                          1
(make-shared v)                                                     2
(remote #T (lambda () (set! v 5)))                                  3
(set! v 9)                                                          4
(remote #T (lambda () (request v) (set! v (+ v 1)) (signal v)))     5
```

Without making **v** a shared variable (in line 2), the assignments to **v** in either remote procedure will not effect its value and the final value of **v**, after all statements have completed, will be 9. With **v** a shared variable, the assignments of each remote will effect the final value of **v**. Depending on the order of evaluation of the assignment statements, which is non-deterministic, **v** can have a final value of either 5, 6 or 10.

The remote procedure call on line 5 requires the shared variable to be locked, using the **request** procedure (see Section 3.2.7), during the assignment because of the multiple references to **v** (once to retrieve the old value of **v** and again to assign the new value). Failure to ensure mutually exclusive access to **v** could result in conflict with other modifications producing invalid values. Access to **v** is

released with the **signal** procedure (see Section 3.2.8). The use of shared variables and send/receive communication is not recommended where more structured forms of communication, such as streams, can be used.

If the shared variable corresponds to a procedure and it is called by a remote task, the procedure call is initiated on the source process. This implements a monitor-style procedure [31], where the procedure can only be accessed by one process at a time, and any closure variables of the procedure refer to the original copies. As such any modifications made to these variables will be observed by all processes. If these variables can only be accessed via the shared procedure, then no explicit locking of these variables is required.

3.2.10 parallel

To implement speculative computation or to synchronise multiple remote tasks, some mechanism is required to cause a task to wait for any computation, from a group of computations, to complete. For speculative computing, it is also expected that the remaining unneeded evaluations will be terminated. The **parallel** procedure implements such a mechanism.

The **parallel** procedure syntax is

(**parallel** *predicate* $expression_1$ $expression_2$ ...)

where $expression_i$ is any valid Scheme^n expression, and *predicate* is a predicate function having one argument. *predicate* should return a true value, when applied to an evaluated expression, only when the **parallel** procedure can complete. The result of the **parallel** procedure will be the value of the expression that passes the predicate test. All remaining evaluating expressions will be terminated (see Chapter 4). If all expressions fail the predicate test **parallel** will return the value of (*predicate* **#F**).

When **parallel** is invoked, arguments are first checked in a left to right order. Checking involves applying the predicate procedure to each evaluated argument. If any argument is an undetermined future, its value is not immediately requested and checking continues with the next argument. If some argument satisfies the predicate condition then no further checking is performed and the argument value is returned as the result.

Once all arguments have been checked, **parallel** will then wait for any of its undetermined future argument evaluations to complete. As newly determined

future values are supplied, each is checked with the predicate procedure. The order of argument checking is therefore indeterminant. The **parallel** procedure will complete as soon as an evaluation satisfying the predicate condition is found or when all future values have been resolved.

Procedures for some commonly used **parallel** predicates have also been explicitly defined. Each of the following procedures can be implemented in terms of the general **parallel** procedure.

(parallel-or *expression*$_1$ *expression*$_2$ ...**)** Completes when any expression evaluates to a true value, returning the value. All remaining undetermined futures will be terminated. **parallel-or** is equivalent to

(parallel (lambda (x) x) *expression*$_1$ *expression*$_2$ ...**)**.

If all *expressions* have been evaluated and all have returned a false result then the **parallel-or** procedure will return **#F**.

parallel-or differs from the standard Scheme logical conjunctive operation, **or**, as arguments are not necessarily checked, in the presence of remote tasks, from left to right. If, however, no undetermined future arguments are present, the **parallel-or** procedure is equivalent to **or**.

(parallel-and *expression*$_1$ *expression*$_2$ ...**)** Completes when any argument expression evaluates to a false value, returning **#F** as the **parallel-and** result. All remaining undetermined futures will be terminated. **parallel-and** is equivalent to

(parallel (lambda (x) (not x)) *expression*$_1$ *expression*$_2$ ...**)**.

If all *expressions* have been evaluated and all have returned a true result then the **parallel-and** procedure will return **#T**.

parallel-and differs from the standard Scheme logical disjunctive operation, **and**, as arguments are not necessarily checked, in the presence of remote tasks, from left to right. If, however, no undetermined future arguments are present, the **parallel-and** procedure is equivalent to **and**.

(parallel-one *expression*$_1$ *expression*$_2$ ...**)** Completes as soon as any one argument is determined returning this as the result, independent of the value. All remaining undetermined futures will be terminated. **parallel-one** is equivalent to

(**parallel** (**lambda** (**x**) **#T**) *expression*$_1$ *expression*$_2$ ...).

(**parallel-all** *expression*$_1$ *expression*$_2$ ...) Completes only when all arguments have been determined. **parallel-all** is equivalent to

(**parallel** (**lambda** (**x**) **#F**) *expression*$_1$ *expression*$_2$ ...).

The result of **parallel-all** procedure is always **#F**.

3.2.11 wait

The **wait** procedure differs from the **parallel** procedure only in the handling of undetermined remote arguments on completion. In the **parallel** procedure, the undetermined remote arguments are terminated when the **parallel** procedure completes as it is assumed their results are no longer required. For the **wait** procedure any undetermined remote arguments are not terminated and continue to evaluate after the **wait** procedure has completed.

The syntax of **wait** is

(**wait** *predicate* *expression*$_1$ *expression*$_2$ ...)

where *expression*$_i$ is any valid Scheme^n expression, and *predicate* is a predicate function having one argument. *predicate* should return a true value, when applied to an evaluated expression, only when the **wait** procedure can complete. The result of the **wait** procedure will be the position index of the expression, with the first expression being 1, that passes the predicate test. If all expressions fail the predicate test **wait** will return the value 0.

The most common predicate functions are defined as separate procedures. They are **wait-or**, **wait-and**, **wait-one** and **wait-all**. These are equivalent to the defined **parallel** procedures, except no remote task evaluations are terminated.

A new constant ***ignore*** has been defined that, when returned as an expression's value, will not be passed to the predicate test. This can be used to ignore certain expression values. In particular, once a value has been returned as the result of the **wait** procedure, it may be set to the value *ignore* so that the same **wait** statement can be reevaluated. For example,

```
(let loop ()
     (case (wait-one a b c d)
```

```
            ('(1) {use a} (set! a *ignore*) (loop))
            ('(2) {use b} (set! b *ignore*) (loop))
            ('(3) {use c} (set! c *ignore*) (loop))
            ('(4) {use d} (set! d *ignore*) (loop))
   ('(0) {all done})
))
```

This code will retrieve each of the values **a**, **b**, **c** and **d** as they become available, using each result then returning to wait for the next until all results have been returned. Without setting a variable to ***ignore*** the same variable would be repeatedly returned from the **wait** procedure.

3.2.12 kill

The **kill** procedure allows the user to prematurely terminate the evaluation of an active remote task. The syntax of **kill** is

(**kill** *expression*)

where *expression* is any valid Scheme^{n} expression.

If expression evaluates to an undetermined future, which has been returned as the initial result of a **remote** procedure, then **kill** will immediately return **#T** otherwise **#F** is returned.

Killing of an undetermined future will cause the associated remote task to evaluate any **on-kill** procedure declared (see Section 3.2.14). The result of this procedure will be assigned as the determined value of the future. If no **on-kill** procedure has been declared the default value of ***ignore*** will be assigned to the future.

Once the **on-kill** procedure has been evaluated the remote task is terminated (see Chapter 4). Termination and the **on-kill** evaluation need not have occurred by the time the **kill** procedure returns. The **on-kill** result is retrieved in a lazy manner, similar to undetermined futures, and need not be supplied until required.

3.2.13 kill-with-no-value

The **kill-with-no-value** procedure is a variation of **kill** which also allows the user to prematurely terminate the evaluation of an active remote task. The syntax of **kill-with-no-value** is

(**kill-with-no-value** *expression*)

where *expression* is any valid Scheme^n expression.

If expression evaluates to an undetermined future, which has been returned as the initial result of a **remote** procedure, then **kill** will immediately return **#T** otherwise **#F** is returned.

kill-with-no-value differs from **kill** in that no value for the undetermined is evaluated by the remote task. No attempt is made to evaluate any **on-kill** procedure declared by the remote task.

3.2.14 on-kill

The syntax of **on-kill** is

(**on-kill** *thunk expression*)

where *thunk* is a no argument procedure and *expression* is any valid Scheme^n expression. The purpose of **on-kill** is to perform relevant clean-up operations when the remote task is prematurely terminated by **kill**. The **on-kill** procedure *thunk* will be evaluated by the remote task when required and the result of this procedure will be returned as the result of the associated future.

3.3 Examples

This section provides some simple programming examples using the constructs defined above. These examples are to serve as illustrations of how the newly defined language constructs can be applied. They are not intended to represent the typical problems or uses for Scheme^n. Chapter 7 develops a more suitable application of Scheme^n.

3.3.1 Example 1: Fibonacci Numbers

The recursive definition of the Fibonacci sequence can be succinctly coded for concurrent evaluation using the code segment of Figure 3.1. Each recursive call of **fib** will generate two further concurrently evaluated recursive calls until **n** reaches the value 1 or 0. Each terminal call will then be evaluated returning a value to the previous procedure call which will in turn be added together and returned. As the + operation is a strict procedure, the result of both sub-components are required for

```
(define (fib n)
  (if (<= n 1)
   {then}  n
   {else}  (+ (remote #T fib (- n 1)) (remote #T fib (- n 2)))
))
```

Figure 3.1: Example 1: Scheme^{n} code for generating the nth Fibonacci number

the procedure to complete. This can quickly lead to an excessive number of remote tasks and, as a practical example, would give rise to poor performance as the remote task overhead exceeds the parallel evaluation gains. Many tasks will be idle, waiting for their sub-components to complete.

To improve the performance of this example the granularity must be set to a level where the number of active parallel tasks is of the same order of magnitude as the number of available processors. This can be done in two ways; firstly, the depth to which the recursive evaluation produces further remote tasks can be limited. This can be done in Scheme^{n} by replacing the remote procedure calls with the following

```
(remote (< max-depth n) fib (- n 1))
```

Remote tasks will be generated until the value of **n** reaches the value of the variable **max-depth**. At this point, further evaluation will be performed by the same task.

The second method for improving performance removes many of the idle tasks from the evaluation by only generating one remote task at each recursive level and using the otherwise idle task to evaluate the other sub-component. Care must be taken to ensure the remote task is initiated before the other recursive call is evaluated. Since the order of evaluation of Scheme^{n} procedure arguments is undefined this requires the remote task be assigned to a variable before the addition is performed.

```
(define (fib n)
  (if (<= n 1)
   {then}  n
   {else} (let (a (remote #T fib (- n 2)))
            (+ (fib (- n 1)) a))
))
```

```
(define (find-root a b N)
  (cond ((= N 0) nil)
        ((= N 1) (solve-root a b)
        (else
          (letrec ((h (/ (- b a) 2))
                   (p (number-of-roots a (+ a h))))
                  (list (remote #T find-root a (+ a h) p)
                        (remote #T find-root (+ a h) b (- N p)))
))))


(define (flatten obj)
  (if (list? obj)
      (append (flatten (car obj)) (flatten (cdr obj)))
      (list obj)
))
```

Figure 3.2: Example2: Scheme^{n} code for finding eigenvalues.

3.3.2 Example 2: Finding Eigenvalues

Problem: Find all the eigenvalues of a symmetric tridiagonal matrix in the interval $[a, b]$.

This can be solved through finding the roots of the characteristic polynomial using multisection and the Sturm sequence property [4]. The method is basically a divide-and-conquer strategy where the interval $[a, b]$ is subdivided using a bisection method dependent on the number of roots within the interval (as determined by the Sturm sequence property) [60]. Subdivision continues until an interval has either exactly one or no roots. The single root within each interval can then be found using a root finding method such as bisection or some more rapidly convergent method. The algorithm can be parallelised by finding the roots of the disjoint intervals concurrently. Figure 3.2 illustrates the use of the **remote** procedure call. Each subinterval can be evaluated in parallel and the results combined together in a list.

The procedure **find-root** decomposes the problem of finding the N distinct roots of the characteristic polynomial into N concurrent rootfinding evaluations on disjoint intervals. Each concurrent task is automatically assigned to an available processor based on the dynamic process load of each processor in the network.

The result of the **find-root** procedure is a list of lists of various depths. The nesting of the lists is dependent on the partitioning required by the algorithm. The elements of these lists comprise the set of N distinct roots and therefore the

```
(define (vector-add X Y)
  (map (lambda (p q) (remote #T + p q)) X Y))
)

(define (matrix-add A B)
  (map (lambda (x y) (remote #T vector-add x y)) A B))
)
```

Figure 3.3: Example 3: Parallel Vector/Matrix Addition

eigenvalues of the tridiagonal matrix.

As the remote invocations immediately return an undetermined future, the root-finding procedure can return before all the roots have been determined. Elements of the list may be temporarily represented as undetermined futures allowing the **find-root** procedure to return while the determination of these roots is being undertaken by remote processes. This allows other processing by the main processor to proceed and even allows the root list to be manipulated by succeeding procedures provided their final values are not required.

To extract the roots from the nested list, the standard list operation, **flatten**, can be applied (see Figure 3.2). Flattening the list forces all elements of the list to become fully determined before the operation can be completed. Implicit touching (see Section 3.2.2) of all elements is required by the **list?** predicate function during the flatten operation. This operation should be performed as late as possible to avoid waiting for results to be determined and maximise the concurrency.

This example serves to illustrate how the **remote** procedure can be used to dynamically partition a problem. Because the **remote** construct introduces parallelism through a single procedure call, very little modification of the sequential version of **find-root** was required to construct the parallel version. The lazy evaluation strategy maximises the parallel activity by relaxing the order of evaluation and so increasing the number of intervals that are searched in parallel.

3.3.3 Example 3: Parallel Vector/Matrix Addition

Figure 3.3 shows parallel Scheme^{n} procedures for vector and matrix addition. Vectors are represented as lists and matrices are represented as lists of lists.

The procedure **vector-add** takes two lists and generates a remote task to evaluate the addition of each element. The immediate result of **vector-add** will be a list of futures that represent the result of the vector addition.

```
(define (make-stream)
  (cons (make-semaphore 0) '())
)

(define (put s v)
  (request s)
  (set-cdr! s (append (cdr s) (list v)))
  (signal s)
  (signal (car s))
)

(define (get s)
  (let ((v '()))
    (request (car s))
    (request s)
    (set! v (cadr s))
    (set-cdr! s (cddr s))
    (signal s)
    v
))
```

Figure 3.4: Example 4: Stream

Two levels of parallelism are taken advantage of in the matrix addition. The first level uses **vector-add** to add the component row or column vectors of two matrices in parallel. The second level uses the parallel vector addition to add each element in parallel. The immediate result of the **matrix-add** procedure will be a list of futures that each represent a component row or column vector addition. Each listed future will represent a list that may consist of further future placeholders representing each resulting element. The matrix addition example therefore has the potential of evaluating n^2 addition tasks in parallel for when calculating the sum of n by n matrices.

3.3.4 Example 4: Streams

As an example of the semaphore and shared variable constructions Figure 3.4 implements a simple stream data structure. The stream, constructed with **make-semaphore**, consists of a semaphore and a list of values. Values are added to the end of the list using the procedure **put** and values are removed from the head of the list using **get**. As elements are added to the stream, the semaphore is incremented using **signal** and decremented with **request** as the elements are removed. The required shared variable exclusive access when modifying the semaphore is ac-

```
(define (dot-product A B)
  (apply + (map (lambda (x y) (remote #T * x y)) A B))
)
```

Figure 3.5: Example 5: Vector Dot Product

```
(define (dot-product A B)
  (let ((m (map (lambda (x y) (remote #T * x y)) A B)))
    (do ((dot 0)
         (w (apply wait (lambda (x) (not (zero? x))) m)
            (apply wait (lambda (x) (not (zero? x))) m)))
        ((zero? w) dot)
        (set! dot (+ dot (list-ref m w)))
        (set-car! (list-tail m w) 0)
)))
```

Figure 3.6: Example 5: Unordered Vector Dot Product

quired in **get** and **put** using the **request** and **signal** procedures.

Figure 3.7 of Example 5 demonstrates how this stream construction can be used for communication between remote tasks (see Section 3.3.5).

3.3.5 Example 5: Vector Dot Product

The vector product, $\cdot$, of two vectors A and B is given by the expression

$$A \cdot B = \sum_i A_i B_i$$

Figure 3.5 gives a parallel implementation that performs the element-wise multiplications in parallel and sums the result.

The summation step of Figure 3.5 is performed by adding the element multiplications in vector order. This may delay evaluation if multiplications complete out of order. As summation is commutative and associative, there is no need to delay the summation waiting for the next vector order multiplication to complete.

Figure 3.6 shows how the **wait** procedure can be used to accumulate the multiplications in an unordered sequence. The predicate of the **wait** procedure checks for non-zero list entries that need to be added to the cumulative dot product. To avoid adding the same element twice the entry, once added, is set to zero so that the predicate test will not succeed. The wait procedure is repeatedly called until all multiplication results have been summed.

```
(define (dot-product A B)
  (let ((s (make-stream)))
    (make-shared s)
    (map (lambda (x y) (remote #T stream* s x y)) A B)
    (do ((i (length (A)) (- i 1))
         (dot 0          (+ dot (get s))))
        ((zero? i) dot))
))

(define (stream* s x y)
  (put s (* x y))
)
```

Figure 3.7: Example 5: Stream Vector Dot Product

An alternative method to summing the dot product multiplications in an 'as-received' order uses the stream construction of Example 4 (see Section 3.3.4) as shown in Figure 3.7. Multiplications are added to the stream by the parallel remote tasks and removed from the stream by the parent task for summation. To use the stream construction as a communication structure between remote tasks, the stream is assigned to a shared variable which is then passed as an argument to each remote task.

Chapter 4

Distributed System Design

4.1 Introduction

This chapter begins by summarising the Scheme^{n} design specification discussed in Chapters 1 and 2. The following section then elaborates the load balancing mechanism of Scheme^{n} including how process workloads are calculated and distributed, and how this information is used to allocate tasks.

The future construct is discussed in the context of providing the required communication and synchronisation mechanisms for Scheme^{n}. The structure of a future data object is described, together with the uses of each future type. The additional referencing required to allow external access to futures and avoid premature reclamation is also presented.

Scheme^{n} provides a mechanism to explicitly terminate remotely evaluating tasks. A number of methods are described for achieving explicit termination and their relative advantages and disadvantages discussed.

4.2 Scheme^{n} Design Choices

To reiterate, the following design choices have been chosen for Scheme^{n}:

Programming Model: Distributed control flow with a large data flow component. See Chapter 1 for details.

Topology: Any statically defined configuration of asynchronous distributed memory processors is supported, ranging from a pipeline to a fully connected network, including non-regular topologies. A Scheme^{n} system consists of a number of Scheme^{n} processes connected in a statically chosen arrangement. The

mapping of the logical Scheme^n topology to the available hardware topology is expected to be provided by the operating system. The Helios operating system is an example of the type of operating system support required (see Chapter 5).

Each process is aware of the total number of processes in the system and how each process can be reached on the available communication network. A *communication distance* is associated with each process. The distance is a measure of the relative time needed to reach a process. In a point-to-point communication network this is most likely to represent the number of intermediate nodes. Processes at a distance of 1 are referred to as *neighbours*. Each process should have at least one neighbouring process.

Granularity: The granularity of the parallel tasks is determined explicitly by the programmer. It is the programmer's responsibility to choose an appropriate granularity for remote tasks based on the algorithm structure and hardware attributes such as processor numbers, topology and communication performance. The number of active concurrent processes should be of the same order of magnitude as the number of processors available and each task should perform enough concurrent work to make the concurrency overhead worthwhile. This indicates a medium to coarse grain is best supported.

Communication: Processes communicate using a remote procedure call mechanism to pass information for an initiated remote task, as arguments of a procedure, and to return the result of the remotely evaluated procedure to the invoking process (see Chapter 5). Both arguments and results can be passed lazily so that the communication of the actual data can be delayed until required (see Section 4.4).

The remote procedure can also access variables in its closure. Shared variables can be used to provide a virtual shared memory interface between the distributed memory processors and to emulate a point-to-point send/receive style of communication.

Synchronisation: The use of futures in remote task communication allows synchronisation to be implicitly defined through data dependencies that exist between the remote evaluation and the process requiring the result. Synchronisation between multiple remote tasks is available using the `parallel` and

`wait` constructs (see Chapter 3).

For shared variables, additional synchronisation is required in the form of semaphores (see Section 4.4).

Partitioning: Using explicit manual task partitioning, each parallel partition is explicitly specified in the program source code as a procedure. The standard Scheme procedure definition syntax is used to delimit task partitions. All built-in Scheme procedures, as well as user-defined procedures can be specified as partitions of the program. Scheme syntax [63], such as **`set!`**, **`if`**, **`and`**, **`or`** and **`define`**, which cannot be **`apply`**'d, cannot be defined as a partition. This does not however preclude their use as statements within partitions.

In order to distinguish concurrent tasks from sequentially evaluated procedures, an alternate method of initiating or calling a procedure as a remote task is required. An additional Scheme procedure, **`remote`**, (see Chapter 3) is used to specify task procedure calls. Procedures initiated with **`remote`** use a remote calling mechanism to determine when and where the task procedure will be evaluated rather than using the standard evaluation mechanism.

Because the remote calling mechanism is initiated during normal procedure evaluation, the instances of **`remote`**, and hence task partitioning, are dynamically determined. For example, the Scheme^{n} statement

`(if (>` *a* **`0) (set!`** *a* **`(remote #T`** *remote-procedure n***`)))`**

will only initiate a remote task to evaluate *remote-procedure* when $a > 0$. This is a runtime determination, making the selection of task partitions dynamic.

Scheduling: Using the standard Scheme evaluation mechanism means that task scheduling in Scheme^{n} is implicitly governed by the evaluation order of Scheme. A parallel task can only begin when the relevant remote procedure call is evaluated during normal sequential processing. Some task scheduling flexibility is possible as task evaluation need not begin immediately on a remote call, and can be delayed until sufficient system resources, in particular processor capacity, are available to accommodate the evaluation. Rejected futures (see Section 4.4) can retain the information necessary to complete a delayed evaluation without forestalling other processing. The data-driven evaluation of futures can implicitly delay an evaluation until the result is required.

Allocation: The allocation of tasks to Scheme^n processes is, by default, accomplished automatically. Allocation is performed dynamically by the Scheme^n system when the task is scheduled for evaluation. To determine which process a task should be assigned, the processing workload of neighbouring processes is examined (see Section 4.3). Tasks are distributed to balance the workload of the processes (see Section 4.3.1).

The automatic allocation is also performed distributively. A task is not allocated to a particular process but is instead distributed in the direction of processes which are estimated to currently have the smallest workload. The user may override the automatic allocation by supplying an explicit location when the remote task is invoked (see Chapter 3).

Once a remote task has been allocated to a process, the evaluation cannot be transferred to another process. The evaluation must continue on the assigned process even if more suitable locations for evaluation become available.

Section 4.3 describes how the workload of processes is defined, how this information is distributed to other processes and the mechanisms used to assign tasks.

Termination: Any evaluating remote task can be explicitly terminated using the `kill` procedure (see Chapter 3). The remote task must cooperate in its own termination by exiting cleanly when a kill message is received. The remote task has the opportunity to define a procedure, using the `on-kill` construct (see Chapter 3), that will be called on termination. This procedure is used to perform any necessary cleanup operations and to return a partial result to the initiating process.

4.3 Load

A Scheme^n process is permitted to evaluate several remote tasks at a time through multitasking. Although no speedup in execution can be directly achieved by multitasking, as only one task is executing at any given time, the evaluation of multiple tasks can provide the process with replacement work when tasks become suspended. The use of multiple remote tasks per Scheme^n process also allow the program to use a finer grain of parallelism. For speculative computing, where the completion of one task can terminate others, allowing an increased range of tasks

to evaluate can result in significant speedup.

A task can be in a suspended state because it is waiting on a future determination or on a semaphore request. These operations do not require any processing time and do not directly affect the processing of other tasks. Tasks which are continuously polling some condition are not in a suspended state and require similar CPU time to tasks which are executing operations required by an algorithm. Continuous polling or *busy-waiting* should therefore be avoided when programming as the evaluation time of other tasks will be affected.

The time required to evaluate a task is proportional to the number of tasks evaluating on that processor. While suspended tasks are not currently consuming CPU cycles, they may be reactivated at some point and since Scheme^{n} does not allow for process migration, a process's commitment to suspended tasks must be considered. Suspended tasks can also affect other remote tasks while it is in a suspended state as resources, that a suspended task may have acquired, can impede active tasks. In particular, the memory requirements of a suspended light weight process (LWP) can be the same as an active LWP and other LWPs may be affected by the reduction in memory. Scheme^{n} factors the potential CPU requirements of suspended LWPs into the estimated workload of the process but does not account for non-CPU requirements of LWPs.

The workload, w, of a Scheme^{n} process is calculated as

$$w = 1000A + 500S$$

where A = number of currently active tasks and S = number of currently suspended tasks. The workload is recalculated whenever changes to the number of active or suspended tasks occur.

The load balancing mechanism attributes a waiting task a load equal to half that of an active task. If no consideration was made for waiting LWPs, a process could be assigned many tasks and become overloaded if all LWPs became active simultaneously. Whether a task will awaken is dependent on the relationship between the tasks which is determined by the user-defined algorithm.

4.3.1 Load Balancing

Because remote tasks in a Scheme^{n} system can be generated dynamically by any process, the pattern of remote task requests can vary considerably between

algorithms. Some algorithms may only generate remote tasks from a single controlling process, while other algorithms may distribute the remote creation equally among all tasks. Under these varying conditions, it is extremely difficult to allocate remote tasks to processes and achieve a perfectly balanced workload on all processes at all points in the evaluation, especially when processes cannot be moved once assigned.

Any load balancing method should ensure tasks are allocated as near as possible to the initiating process. When a task is allocated, the information required by the task must be transferred to the selected process and on completion of the task the results must be returned. It is therefore preferable if the task is allocated to a process "near" the initiating process so that communication overheads can be minimised.

To provide these facilities for the allocation of an undetermined number of remote tasks, each of which can take an indeterminate amount of time to complete, Scheme^{n} uses a heuristic method. The method assumes future task creation is equally distributed among processes and occurs at regular intervals and that completion times of all tasks are the same. With these assumptions, task assignment can be based on the current workloads of the processes and the communication distance from the initiating process. Although the communication distances between processes are fixed, the workloads of processes will change dynamically as remote tasks are initiated, completed or suspended.

To determine the 'best' allocation, up-to-date information about all processes must be provided. In a distributed memory system, informing all processes of current workloads would involve either broadcasting workloads to all processes whenever a change to the workload occurs, or requesting the current workload from all processes whenever a task allocation is performed. Either method involves a considerable amount of message traffic which does not scale well when processor numbers increase. Even if a more centralised approach to allocation is used, where a single process is sent all remote task requests and assigns them, hence requiring processes to send workload information to one process only, scaling can cause a bottleneck in the sole allocation process.

Much of the workload information is seldom important to a specific allocation, as distant processors would rarely be assigned a task as task locality is preferred. Also, to ensure the information received is exact, no change to any process's workload can occur when an allocation decision is in progress. This requires

a global synchronisation of all processes to suspend processing until any allocation decision has been reached. Even after the decision has been made, any change in the workloads can invalidate its claim to being the 'best' solution. The overhead in providing a 'best' assignment solution is therefore not warranted and a 'near-best' solution is adequate.

In a 'near-best' solution no global synchronisation is required and the last workload information received from a process is assumed to be its current value. In accordance with Scheme^{n}'s distributed design philosophy, a centralised approach has also been disregarded. Scheme^{n} uses a distributed load balancing allocation algorithm that both avoids the need for broadcasting workload messages and the potential bottleneck of centralisation. The distributed solution gives each process the choice of evaluating the remote task, or suggesting a better alternative process. The choice of an alternate location is further limited to a neighbouring process only. It is then the neighbour's responsibility to evaluate the remote or find a better location from its set of neighbouring processes.

4.3.2 Load Estimates

The suitability of a process for evaluating a remote task is based on its current load and the load estimates of neighbouring processes. The *load estimate* of a process is related to both its own current workload and to the load estimates of its neighbour processes. The load estimate, e, is calculated by each process as

$$e = \frac{\sum_{i=1}^{n} e_i + 2w}{n + 2}$$

where e_i is the workload for the ith neighbouring process, w is the process's own workload, and n is the number of neighbouring processes. It is the load estimate of a process which is transmitted to neighbouring processes and not the workload itself.

When a new remote task is received by a process, either from some other process or by one of its own running LWPs, the process compares the load estimates of neighbouring processes with its own workload to determine the best location for a new remote task to be allocated. If its own workload is less than or equal to the load estimates of all neighbouring processes then the remote task will be evaluated locally. If another process has a lower load estimate than the local workload then the process will transfer the remote task request to that process. A process receiving

a remote task request will apply the same logic to determine if itself or another process it is connected to is best suited to evaluating the task.

The allocation decision can also decide to reject a remote task request. Rejection of a task occurs when a process and all its neighbours have a workload estimate or load that exceeds some fixed limit. For the current implementation (see Chapter 5), a workload limit of 2000 has been chosen. This represents a maximum of four suspended LWPs per process. It is therefore possible, with a maximum load of 2000, to achieve a load of 4000 if all previously suspended LWPs become active.

A high load on a set of neighbouring processes suggests that the load of all processes is high. Continuing to search for a suitable location in such a circumstance has only a small chance of succeeding. The allocation process is therefore stopped and a rejection message returned to the initiating process. On reception of a rejection message, a process can immediately re-determine an alternate remote task location, evaluate the remote itself, or delay the remote evaluation until either the result is required or until the process loads are reduced (see Section 4.4).

The accuracy of the workload estimates will depend on how often this information is updated and transmitted to other processes. If workload estimates are transmitted to other processes whenever a change occurs, the effect on workload estimates will propagate throughout the system. In fact, because a change in a process's workload will effect a neighbour's workload estimate, which will in turn effect its own workload, which will again effect its neighbour's, the number of messages sent can be high. The workload estimates will eventually reach a steady state, as the size of the change in workload is decreasing, but the number of messages sent makes this approach unsatisfactory.

To avoid this behaviour, a process will only transmit a new workload estimate when a change occurs in its own local workload due to LWP creation, termination, or suspension. Changes in workload estimates will therefore only propagate as far as neighbouring processes, which will update their own workload estimates but will not transmit the updated value to other processes, including the process which induced the change. This restriction makes workload estimates less accurate but avoids excessive message passing.

Several methods can be employed to improve a workload estimate's accuracy without requiring any more communication. Whenever any Scheme^n message is transmitted, an up-to-date workload estimate is included in the message as part of a fixed sized message header (see Chapter 5). Also, when a remote task request

is passed to another process for evaluation, the sending process assumes the receiving process will evaluate the remote task. Thus the sending process can update its workload estimate based on this assumption without need for any additional communication. The modified estimate can be sent with the remote task request message to improve the receiving process's workload estimate. If the receiving process does not evaluate the process, but sends it to another process for evaluation, an additional message is required by the remote evaluation mechanism to determine the final destination of the remote (see Section 4.5). This destination message will provide a corrected workload estimate at no additional cost.

In this manner tasks may be distributed throughout the network as a result of a number of locally made decisions by processes as to the relative workloads of other processes. No single process is responsible for the assignment of remote tasks and each process is capable of creating, distributing and evaluating a remote task. Remote task requests are sent to nearest processors first but can be distributed to any process in the network.

4.4 Future Structure

A future object consists of a future id, a process id, a future type and a value field. The user does not have direct access to any of this information. The id of a future is the same as the id of the associated remote request and is used to match the remote reply message to the correct future. When a remote task has completed, a remote reply message is returned to the source process. The message contains the result of the remote task evaluation which is assigned to the value field of the future structure. The value field is then referenced when the future is touched. The process id field of a future contains the id of the process on which the associated remote task is being evaluated. The `kill` procedure uses this id to send a kill message to the relevant process. The future type field indicates the status of the future. The following types are used for futures:

- Undetermined
- Determined
- Terminated
- Terminated-with-no-value

- Rejected
- Lazy
- Shared
- Error.

Undetermined: Initially, when a **remote** procedure creates the future structure the future is given an *undetermined* type. Referencing an undetermined future will suspend a LWP until a value has been determined by the associated remote task.

An undetermined future has no value but the value field can be used to store a copy of the procedure and arguments passed to the **remote** procedure call. A copy of this information is retained by the undetermined future to allow the source process to evaluate the remote request itself if necessary. If the local task touches the undetermined future object and the associated remote request is still present on the local remote handler queue then the queue entry is removed and the local task evaluates the remote task in-line. Using an existing task to evaluate a required future that has not yet begun evaluating will avoid overhead and reduce workload. By using the local task no additional LWP need be created and no messages generated. The local task which would be forced to wait for the remote request anyway can be better utilised by evaluating the information it requires. It can then continue in the shortest possible time.

Situations where a remote request has not begun to be processed and is still on the local remote handler queue can arise when the result of a remote evaluation is required soon after the **remote** call was given or when the workload of the local process is high and other remote requests require processing. If the remote request has already begun to be processed then touching an undetermined future will suspend the touching task. The task will be awakened when the future becomes determined.

Determined: When a remote request has been satisfied the local future handler process receives a remote reply message from the remote task (see Chapter 5). The future handler matches the remote reply id with entries in a future table (see Section 4.4.1). The future table entry, created by the **remote** procedure, contains a reference to the future structure. The future handler replaces the undetermined future found with a *determined* future whose value field now contains the returned

result. The future handler will awaken any LWP waiting for a remote reply to be returned so it can continue processing.

The future handler will also send future reply messages to processes which have requested the value of futures which were undetermined at the time of the request. The list of these processors is stored by the future handler when it could not satisfy the original request.

Rejected: In situations where no suitable process could be found to process the remote request, a remote rejection message is returned to the source process. These messages are handled by the future handler by finding the associated future structure using the future table and message id and changing the type of the future from undetermined to *rejected.* A rejected future is therefore one for which no value has been determined and is not actively being calculated. When a local task requires the value of a rejected future it is necessary for the local task to evaluate it itself. The rejected future still retains the procedure and arguments of the remote request in the value field, as assigned at creation, so the local task can evaluate these to produce a result. The user remains unaware of whether a future has been determined or rejected with the same result being produced by each but with possibly different performance characteristics.

A drawback of the rejection method is that no attempt is made to retry the rejected remote request at a later date. Although no suitable process was found to evaluate the remote request initially, changes in the workload of processes between the time of rejection and the time at which the result is required may allow the remote request to be satisfied. The rejection of a remote request therefore removes any chance of concurrency and forces the initiating task to evaluate the remote request albeit at a later time. One solution to this is to reinstate the remote request of rejected remotes periodically. A suitable time to attempt this would be during garbage collection. At garbage collection the future table could be searched for futures with a rejected type. A remote request can then be recreated from the information held in the rejected future and queued to the remote handler for processing. Checking for rejected futures can be avoided if the workload of the system is currently high as it is likely they will only be rejected again. Care must be taken to retain the reference to a rejected future in the future table until it is evaluated or not required to avoid premature garbage collection.

Terminated: When a local task no longer requires the result of a remote task previously initiated, the remote task may be terminated by performing a `kill` operation on the associated future. Killing a future changes the future type to *terminated* and generates a kill message which is sent to the process id noted in the future structure as the destination process of the remote task. The `kill` procedure does not wait for confirmation of the termination and returns control immediately to the local task.

When a kill message is received by the remote task it is given the opportunity to return a result in a remote reply message before it finally terminates. It is sometimes important to note the status of a terminated remote task. This status can be used to continue the processing of a task at a later date or as information for some other task. If a local task touches a terminated future the local task will be suspended until the terminated remote task returns a result. When a remote reply is received by the future handler for a remote that is already terminated, the future type is changed to determined and the returned value is assigned to the value field. Any LWP suspended on a future will then be awakened.

A terminated remote task can return a value either from successfully completing before the terminate command was actioned or by returning a partial result as a consequence of the kill message. It is also possible for the remote task to be terminated before it could begin. To avoid a LWP from being permanently suspended waiting for a reply from a remote task that had no opportunity to return a reply, the Scheme^{n} LWPs will return a default reply of false for the remote in this situation. It is the user's responsibility to ensure that the three distinct results that can be returned by a terminated remote can be distinguished. In other words, if a result from a terminated remote is required then as the system returns false by default a false value should not be returned by the successful evaluation and only returned as the result of termination to indicate no progress was made. Similarly, the value returned through forced termination should be kept distinct from values that are returned on successful completion.

Terminated-with-no-value: A future can be in this state after a `kill-with--no-value` procedure has terminated the associated remote. Because no value has been requested by the explicit termination, no value is associated with this future. Read access to futures in this state should not occur and an implementation is free to handle such access as necessary. Possible responses to such access include generating an error or returning a predefined value such as `#F`.

Lazy: By default remote procedure arguments are transferred by the `remote` procedure in an eager fashion. All arguments are converted for transmission and sent together with the remote procedure to another process.

Rather than transferring a potentially large data structure between processors, where it is possible that it will not be required, a *lazy future* can be used to delay the sending until it becomes necessary. By specifying arguments using the `lazy` procedure a typically much smaller lazy future will be transferred instead.

When the value of a lazy future is required by the remote process, either explicit or implicit touching of the future will transfer the actual value of the lazy future from the source process. Lazy futures have the same property as other futures that only strict references require the value of the future. Lazy futures may be manipulated as function arguments and assigned without the need for their actual values. Lazy futures received from a process can be passed as arguments to other remote procedure calls, as well as returned as the result of the remote procedure.

The transfer of the lazy future's value is initiated by the remote process by sending a *future-request* message to the source process, and then converting the remote lazy future to an undetermined future type. The `touch` procedure will then wait for the future value to be supplied. This differs from the touching of undetermined futures, as a request for the value to be supplied is needed before the value will be sent. Once the request has been made however, the future is equivalent to an undetermined future.

To avoid waiting for the lazy future value to be supplied once it is known that the value is required, an explicit request can be initiated before the value is needed. The procedure (`force` *lazy-var*) will send a future-request message and convert the lazy future to an undetermined future type, but unlike the `touch` procedure, `force` does not wait for the future value to be supplied. When the value is truly required, either it will already be available, as the value of a determined future, or the touching will need to wait on the undetermined future until the value is supplied. The `force` procedure can also be called by the source task to immediately send the associated value. This may be necessary if the lazy variable is to be modified destructively.

If the source process reassigns the value of a variable that has been passed as a lazy future but has not yet been transmitted, the lazy future will be unaffected and will still have the previous value. This maintains the expected evaluation order among the concurrent processes and the semantics of the program is unaffected

by the presence of the lazy future construct. For destructive operations that can mutate the value of an unsent lazy future, such as `set-car!` and `vector-set!`, the unmodified value must be sent to the remote procedure before changes are made. This is necessary to ensure that the result of a remote procedure that uses lazy futures is unaffected by exactly when the lazy future is transferred.

The value associated with a lazy remote need only be transferred once from the source process. Succeeding references to the future by the remote process will use the same value as that retrieved by the first reference. As with normally transferred arguments, any modifications by a remote procedure to the value of a lazy future, either by assignment or a destructive operation, will not affect the value of that variable anywhere other than within the remote procedure itself.

The lazy procedure is also used implicitly when other categories of futures are to be sent as arguments. If the lazy procedure is applied to an already lazy future or a shared future then no operation is performed. If a determined future is to be made lazy, it is the determined value of the future that will be sent on a lazy request. Undetermined, terminated and rejected future types are all sent as lazy futures. When a remote process requires the value of a passed future, a request is made for the value of the lazy future. If between the time of the sending of the lazy future to the remote process and the receipt of the lazy request by the source process, the future has become determined then the source process will furnish the determined value to the remote process. If, however, the future remains unevaluated then no value will be immediately supplied as a result of the lazy request and the future table entry on the source process will be marked to indicate the value has been requested. When the result of the associated remote evaluation is returned to the source process, the determined future value will be sent to the marked remote process. The use of lazy futures for distributing unevaluated futures ensures that changes to the status of the future on the source process will be recognised by the remote processes.

Lazy futures can be viewed as a form of distributed memory pointer. These pointers can be used to reference data in external distributed memory spaces. They allow a form of read-only virtual shared memory to be implemented. Because of the read-only property of lazy futures, no synchronisation is required between the lazy future references. Any changes made to a lazy variable by a process is only visible to that process as only that copy of the lazy variable is affected.

Shared: Unlike lazy variables, where multiple copies of the variable's value can exist, the shared variable is restricted to having a single copy existing on only one processor. Processes that wish to read or modify the variable must do so via the owning process. Whenever a process requires the value of a shared variable it implicitly or explicitly requests the value from the owning process in the same manner as lazy futures. The returned value is then used by the remote process but is not recorded as the value of the future. The next reference to the shared variable will also require communication with the owning process. In this manner, only one value for the variable is maintained.

To assign a value to a shared variable, the standard set! operator is used. When the variable to be assigned is a shared variable, the set! operator passes the value to the owning process in a similar manner to the replied value of remote procedure calls. The passed value is then assigned to the shared variable in an atomic operation.

If more than one operation is to be applied to a shared variable by a process, access to that variable should be denied to other processes during this critical period. Mutual exclusion is invoked on a shared variable using the procedure (**request** *shared-var*). The **request** procedure makes a request of the owning process for exclusive access to the shared variable. When the owning process grants exclusive access, the requesting process is supplied with the current value of the shared variable. All further accesses to the shared variable can then be carried out locally. Other processes are denied both read and write access while the variable is locked.

When the process has completed its critical section operation on the shared variable it performs a (**signal** *shared-var*) call which returns the latest value of the variable to the owning process. The owning process is then free to grant read and write access to other processors again, satisfying any requests that were blocked from occurring while the variable was locked. All read accesses will be granted first then the first process on the list of lock requests will be given mutually exclusive access. All non-locked write accesses will be ignored whenever the shared variable is locked. Any change in value that a single write access has on a shared variable will be immediately replaced by the result of the locked access, so there is no need to action the write.

When a shared variable is used in a non-strict operation, such as argument passing or list concatenation, the shared future, like all future types, does not require the value of the shared variable. At some later stage, the value will be retrieved but

this will be the latest value of the shared variable which may have changed. If the value of the shared variable when the non-strict function is applied is required, the **touch** procedure can be used.

Error: The error handling mechanism of Scheme^n has been incorporated into the future structure through the *error* future. Any remote task which causes an error returns, as the result of the task, an error future. This contains information about the error that can be handled by the calling task. If the value of a remote is not required the error state of the associated future is not important and processing continues normally. If, however, the result of a failed remote task is required, the access to an error future causes the accessing task to fail. In this manner, errors are propagated back to the initial process of the evaluation.

4.4.1 Future Table

When a future is passed to another processor, a record of the communication is made. The reference is primarily used to correctly handle the garbage collection of the distributed future objects. While objects that are local to a processor can be garbage collected independently to other processors, when no local references to that object exist, objects that can be referenced by other processors cannot be garbage collected until all external references are removed as well.

The *future table* is used to maintain a list of processors that have been sent a reference to a future and hence can request access to that future. The future table reference is sufficient at garbage collection time to ensure that a future, that is not otherwise locally referenced, is not prematurely disposed of. When remote processes perform garbage collection of futures for which they are not responsible, they return a message to the source process to acknowledge this. The source process can then remove the reference to that processor from the future table and on the next garbage collection remove the future if no other references exist.

When lazy futures are distributed by a process other than the source of the future, a Scheme^n implementation can use one of two methods for recording the transfer. The first method centralises the responsibility for the future with the process that generated it rather than the sending or receiving process. When a non-local future is passed to another process, the sending process must also inform the future source process of the transfer so that it is aware of another potential referencing source. Once the intermediary process has informed the source , no fur-

ther responsibility for the future is assumed. No record is kept, by either sending or receiving processes, of the source or destination of a non-local lazy future transmission. All processes which access the future will query the source process directly when the value is requested and inform the source process when the future is no longer required.

The second method for sending non-local lazy futures maintains a distributed view of those processors that can access a future. Whenever a process passes a future to another process, whether it is the source process, or not, it records a reference of the transfer in its future table. No direct communication is made to the source process because of the transfer. The receiving process is aware of both the origin of the future and of the process from which it was supplied the future reference. These details are recorded in the source and destination fields of the future data structure. A process may then request the value of a future from either process. The choice of process to request from is implementation dependent and could be chosen dynamically, for example based on relative communication distances or process workloads, or statically where information is always requested from the same process. If a request is made to a process which does not have the value, that process must request the value from its source and relay the value to the requesting process. During the relaying of a request the intermediate process may retain a copy of the information thus avoiding duplication of requests. When a process no longer requires a value, in the second method, it informs the process that supplied it with the future rather than the future source process. The supplying process can garbage collect its reference to the future when no local or future table references exist.

Either of these two methods can be used or a combination of both can be employed where new references to a future are relayed back to the originating process as well as being recorded by the sending process.

The future table is also used to record external requests for values that could not be satisfied immediately. When a value for a future becomes available all external references to that future are checked in the future table. Any references marked for a request are immediately sent the value.

4.4.2 Future Checking

In order to implicitly suspend Scheme operations on the evaluation of undetermined futures and to be able to replace determined futures automatically with their equivalent values, the Scheme^n implementation must check the type of each

argument to determine if it is or is not a future before using it. If it is a future, it must be replaced by its equivalent value. For determined futures, this requires the reference to the future object be replaced with a reference to the future's value field. For undetermined futures, the local task must be suspended until the value becomes available. The newly determined future can then be replaced by its value.

When the Scheme^n implementation is performing the standard type checks on procedure arguments the implicit future checking does not add any additional overhead to the evaluation of non-future arguments. If an argument is of the correct type then it cannot be a future so no additional future checking need be done. If an argument is not of the correct type then a future check is performed. If the argument is a future its true value is determined and the type checking is reapplied. For arguments which fail the argument type check and are not futures the procedure will return the appropriate type error. Assuming valid arguments, future checking is therefore only performed when a future is actually present.

When no standard type checking is being performed future checks are still required to identify future arguments. The need to check each argument for futures can add a significant overhead to the performance of each and every built-in Scheme operation. The overhead can be reduced as most of the future checks will be redundant. Once an object has been checked and identified as a non-future object then it need not be checked again until its value is reassigned. To determine which future checks are required and which are not requires functional analysis to be carried out.

Alternately the implicit future checks can be removed altogether and the user can be made responsible for inserting future checks at the appropriate places. To ensure an object is not a future the user can explicitly call the `touch` procedure on each suspect object to convert them from futures into matching values. The `touch` procedure will force the local task to wait until the object has been evaluated in the cases where the object is an undetermined future, and will replace the reference to the future with a reference to the future's value. Touching a non-future object has no effect.

4.4.3 Future Garbage Collection

During garbage collection all references to determined futures can be replaced by references to its actual value specified in the value field. This may occur during remote task evaluation or when the remote task has either successfully completed or been prematurely terminated without requiring the future's value. If an

undetermined future is garbage collected the associated remote task should be terminated. The garbage collection algorithm should send the required kill message, specifying that no return value is required, to free up the computing resources being used.

4.5 Explicit Termination

Because of the specifics of the Scheme^n implementation environment, it has not been assumed that a LWP can be terminated by other LWPs. The need to terminate unwanted remote tasks has already been discussed (see Chapter 2) and the procedure `kill` has been included in the Scheme^n language design (see Chapter 3). To implement the `kill` procedure in an environment where the exception handling capabilities do not permit external termination of LWPs requires the LWP to be killed to actively cooperate in the termination process. If a LWP is informed that it is no longer required it can stop processing and release its computing resources.

A LWP process can be given information through a shared global variable but it also requires the LWP to regularly check for this termination condition. Some possible methods of a LWP recognising that it should terminate while still performing general computation include:

- Regularly timed interrupts.
- Explicitly added condition testing in an algorithm's code.
- Inclusion of termination tests in commonly used code.

If the computer hardware supports the generation of interrupts on a LWP basis then regular testing for termination can be performed with no special consideration by the main algorithm. The interrupt mechanism must be executed as part of the same LWP so that the LWP can terminate itself. The design of Scheme^n does not assume interrupt capability is available and must use an alternate approach.

Without hardware support it is difficult to ensure an algorithm will routinely check for termination. During general processing, an algorithm can be evaluating a variety of different operations which must be interspersed with termination checking code. The frequency of checking is therefore dependent on the sequences of operations in an algorithm. Adding the correct number of checks at evenly spaced intervals is important for good performance but is difficult to achieve. Too many

checks result in wasted CPU cycles while too few may waste processor resources by not terminating a LWP as soon as possible. Termination checks can be automatically inserted during compilation but determining an adequate placement of checks is computationally intensive and may be insolvable statically.

For example, consider a loop with a large number of iterations where each iteration can be computed quickly. Inserting checks before and after the loop only may not provide enough checking but performing a check for each iteration may cause too much checking to occur.

An alternative to inserting checks for termination into the algorithm itself is to only insert checks into system code. Provided the system subroutines are commonly accessed by the algorithm, checking will occur frequently. This solution is simpler than inserting checks within an algorithm as no changes are required to the algorithm itself. The problems of frequency and regularity still exist however as when system routines are evaluated is highly dependent on the algorithm. This is the method chosen for the prototype Scheme^n implementation.

4.5.1 The Kill Message

To terminate a remote task the source process must send a kill message to the process on which the remote task is being evaluated. Because of the distributed manner in which remote requests are allocated to processes the source process does not assign remote requests explicitly to a specific process and is therefore not aware of the final destination of a remote request. To identify a remote task's destination therefore requires additional information to be specified and recorded.

It is desirable that a kill message reach the destination process as soon as possible for efficiency but it does not invalidate the result if the kill message is not received and the remote task is not terminated. The additional overhead introduced to implement killing should also be minimised. In a sense, the need to terminate a remote task is counter-productive as the resources spent on the remote task up until that point are wasted. For efficiency, the necessity for killing should seldom occur as terminated remote tasks do not make effective use of computing resources. And as remote task termination is not necessary for an algorithm to correctly evaluate, any performance gains achieved through killing must be carefully weighed against the overheads and required complications introduced to the parallel processing system.

To implement remote task termination, several methods of finding and informing the destination process of termination can be used, including:

1. Each redistribution of a remote request by intermediate processes is remembered by the intermediate process so that any kill messages received can be redirected.

   Problem When can an intermediate process forget about a redirected process.

2. Broadcast the kill message to all processes.

   Problem Broadcasting is inefficient for large networks of processors.

3. The destination process of a remote request returns a message to the source process identifying the process id of the destination process.

   Problem Every remote task requires additional messages to be sent.

Each of the methods suggested above require either additional messages or additional information between processes. Because of the design philosophy of Scheme^n, the number of messages required should be minimised as the expected overheads involved in communication are significant. Information required by each process about the status of other processes should also be kept to a minimum to reduce the dependencies between processes.

Method 1: The first method suggested requires intermediate processes to be involved in the task activities of other processes. The only relationship between a remote task and an intermediate process is that the intermediate process was used to distribute the original remote request. Maintaining a reference to a redistributed remote task is potentially dangerous as it creates links between processes that are not directly involved with the initiation, evaluation or termination of a remote task. When the status of a remote task changes, intermediate processes must be involved in ensuring the consistency of state information.

In particular, when a remote task has completed, either normally or prematurely, references to the remote task must be removed from all intermediate processes. This could be done as the remote task returns a result to the source process, but it requires the return path of a remote task's result to exactly match the path used to distribute the remote request. This implies the links maintained between the source and destination processes must be bi-directional to allow messages to be sent from both the source and destination. To minimise the communication overhead involved with remote task evaluation, results of a task should be returned by

the most direct route. But as the distribution path of a remote request was chosen dynamically, as a consequence of process workload estimates, it may not represent an efficient path for communication between the source and destination processes.

The distribution path of a remote request can be quite convoluted and can contain cycles caused by a remote request being returned to a process which has already distributed that same request. Cyclic paths further complicate the links required between source and destination processes as multiple references to the same remote task can now occur. The cycles created can either be removed from the path, requiring further communication between processes, or a more complicated algorithm can be used by each intermediate process to access the multiple references.

Method 2: The second method suggested for remote task termination requires no knowledge of the destination of a remote task either by intermediate processes involved in the distribution, or by the source process itself. By broadcasting the kill message to every process it can be guaranteed that the message will be received by the process evaluating the remote task. In situations where a remote task is not prematurely terminated and no kill messages are sent this method does not incur any overhead. But the overhead incurred to terminate a single remote task is significant as it is incurred by every process in the system. For communications which involve only two processes, in this case, the source and destination of a remote request, the use of broadcasting is inefficient. When many processes can broadcast at once system-wide communication can seriously effect a system's performance especially for large distributed multiprocessor systems. Depending on the broadcast procedure, some messages can be avoided if, during the broadcast process, the destination of the relevant remote is identified as no further broadcasting is required from this point.

Method 3: The third method for terminating remote tasks has the destination process inform the source process of a remote task's destination. The destination message, which has the same message id as the remote request and specifies the destination process id, is sent to the remote request's source process to inform it of the final destination of the remote task. As the source process is the only process that can terminate a remote task it is the only process that requires information of a remote task's destination. When a remote task is to be terminated the termination can be accomplished simply by sending a single message directly to the destination process. The drawback of this method is the need to send a message from a remote

task's destination to its source for every remote task initiated. Assuming killing occurs infrequently, most destination messages are not required but are sent anyway.

There is a period between the initiation of a remote request and the reception of the remote destination message during which the source does not know a remote task's destination and cannot send an appropriate kill message. If the destination is unknown when a terminate command is invoked, the kill message must be delayed until the destination is determined. To avoid delaying a parent LWP wishing to terminate a child LWP the parent can set the type of the relevant future to terminated and assume the remote task will be terminated when possible. When the destination message arrives and the associated future has been terminated the source process can then send the kill message to the destination process to perform the termination without any additional interaction with the parent LWP.

To help reduce the number of destination messages required and to avoid the delay while a destination is unknown, the third method can estimate the destination of a remote task. By initially using a default destination for a remote task, the source process can immediately send a kill message when the true destination has not been received. If the destination assumption was correct then the delay is avoided. If the assumption was wrong the arrival of a destination message for an associated future that has been terminated can be used to force the source process to resend a kill message to the true destination process.

Because a kill message can now be sent before the destination process has sent a destination message, it is possible a remote task to be terminated has not as yet begun processing. The remote request may still be pending on the remote handler queue of the destination process waiting to be processed. The process receiving a kill message must therefore check its remote handler queue for the remote request so that it is not overlooked and evaluated by the process after the kill message has been processed.

When a remote is specified by the `remote` procedure, it is first placed on the source processes remote handler input queue. When killing a remote task and a true destination has not been designated, the source process should therefore first check its own remote handler queue.

The process which receives a kill message for a remote task that it is not currently evaluating and which is not on its remote handler input queue can simply ignore the kill request as the source will be prompted to send another kill message by the true destination process. Another situation where a process may receive a

kill message for an unknown remote task is when the kill message arrives after the remote task has successfully completed. Even when the destination is correct, it is possible that the remote task that was to be terminated may have already completed naturally and is at present returning a result to its source. In this situation the destination process will also not find any record of a remote task with the specified id. In this situation the destination process can also safely ignore the kill request.

By selecting the default destination of a remote request to be the process to which the source process initially sends the request no destination message needs to be returned by this neighbouring process if it is the true destination. This is not necessarily the case as the neighbouring process also load balances the remote request and may distribute the message further. This may happen particularly when the workload estimates of processes used to distribute the remote request are incorrect. If the workload estimates are up-to-date then it is likely that the neighbouring process will evaluate the remote task so making the assumption of the destination can significantly reduce the number of destination messages that need to be sent. A process knows it need not send a destination message if the remote request is received directly from the source process. Care must be taken to reset the initial destination if a remote request is returned to the source through the standard load balancing procedure and the source then redistributes it.

Chosen Scheme^n method: Of these methods, the third approach has been chosen for the implementation of Scheme^n. Its main advantages are the need to only involve the source and destination processes of a remote evaluation in the remote tasks termination and the low number of additional messages required. Knowledge of a remote tasks destination can be of use to the source process in analysing system workloads.

In many situations, the status of terminated remote tasks do not provide any useful information and it is only necessary to terminate the task to free the computing resources used. An alternate terminate procedure, **kill-with-no-value**, is provided to avoid generating results and remote reply messages for results that will not be used. The **kill-with-no-value** procedure sets the future type to terminated-with-no-value and generates a kill message to be sent to the destination process as with the **kill** procedure but the message indicates that no remote reply is required. The **on-kill** procedure is not called by the destination process when a **kill-with-no-value** kill message is received.

As well as the **kill** procedure, remote tasks can be implicitly terminated from within a **parallel-or**, **parallel-and** or **parallel-one** procedure and during garbage collection. For each procedure **parallel-or**, **parallel-and** and **parallel-one**, any undetermined future arguments remaining after the procedure can complete will be implicitly terminated. As with other Scheme data objects an undetermined futures may be garbage collected if all references to it are removed. If an undetermined future is garbage collected this implies the result of the associated remote task will not be used once it is evaluated. The garbage collection algorithm should therefore send the required kill message, specifying that no return value is required, to free up the computing resources being used.

When a task is terminated and it has generated remote requests that have not been satisfied the child tasks should also be terminated unless the results are globally accessible. If the results of child tasks are not required the associated undetermined future will naturally be garbage collected. Child tasks of terminated tasks will therefore be automatically terminated if there results are no longer required.

Chapter 5

Implementation

5.1 Introduction

This chapter describes the implementation environment for a prototype Scheme^n implementation. Both the hardware and software components of the implementation are presented. To provide the Scheme^n model of communication for a distributed memory passing environment, some form of message passing protocol is required. The Scheme^n message passing protocol is defined for distributing information including Scheme^n data, remote tasks, and workload estimates. The effects of copying Scheme^n data on the structure of the data are also discussed.

The chapter continues by describing the set of light-weight processes required by each Scheme^n process in the network. The LWPs are responsible for communication and synchronisation. They allow communication between processors to be asynchronous and be received from processors in an undetermined order. They also provide the synchronisation between processes as well as the synchronisation of access to local shared resources such as memory and I/O.

LWPs are required to receive messages from each communication channel. These messages are queued to either the remote handler or the future handler for further processing. The remote handler is responsible for parallel task creation requests, while the future handler is responsible for the returned evaluation results.

5.2 Implementation Environment

This section describes the hardware and software environment used to implement a prototype Scheme^n system. In conjunction with a concurrent language's

ability to represent parallelism, a computer system must also contain the underlying mechanisms to support the concurrent evaluation. Mechanisms for communication, synchronisation and process control should be provided by both the hardware and software.

The hardware available for the implementation, was a board of Transputers [35] connected to a standard IBM compatible personal computer. The Transputer system was not run as a back-end platform, where all development is cross-compiled from the front-end processor, but instead the operating system Helios [54] was used on the Transputers themselves to provide a standard Unix-like environment for development. Using these system components, the public domain implementation of Scheme, Scheme->C [6], was ported to the Transputer/Helios environment, from which modifications where made to produce a working Scheme^{n} implementation.

5.2.1 The Processor: Transputer

Each Transputer [25], manufactured by Inmos, is a 32-bit RISC-architecture processor with an on-chip floating point unit. There is explicit support for multiple processes and multiple processor systems. Transputers are connected together through high-speed communications links that provide point-to-point interprocessor connections. There are four serial links per Transputer for full duplex, synchronous communication. Each link has its own DMA (Direct Memory Access) controller allowing independent and parallel operation with the other links, the CPU and the floating point unit.

The implementation hardware consisted of nineteen 25MHz T805 Transputers each with 4 MBytes of memory and a link speed of 20 MBits/second. A 20 MBits/second link speed corresponds to uni and bi-directional data rates of 1740 and 2350 Kbytes/second respectively. The processors were arranged in a ring of sixteen processors with three additional processors used for user login. The processing speed of each of these Transputers is roughly ten to fifteen times slower than a Sparcstation 2, model 4/75.

5.2.2 The Operating System: Helios

Helios [54] is an operating system available on a range of distributed memory machines, including the Transputers. Other software environments for the Transputers include Express, Trollius, Mach, and Meikos [25]. Helios is Unix-like in

appearance and has a high compatibility with Unix. A pipe mechanism, similar to Unix pipes [71], is provided for synchronous point-to-point process communication.

Helios uses a logical description of an application's process components and their connections to automatically map these processes onto the available processors and communication channels, including the possibility of allocating more than one process to a single processor. Each process component is an independent executable program that uses pipes to communicate with other components. The automatic and dynamic mapping of processes to processors by Helios allows an application to run independent of physical network properties.

Each Scheme^n process consists of a Scheme^n core program with optional user-defined procedures compiled and linked into each. In the current implementation each process must be identical because of the method used to reference remote procedures (see Section 5.3). The dynamic generation of remote tasks is achieved using the system procedure, **Fork** [54], to create multitasking LWPs within the appropriate Scheme^n process environment.

5.2.3 The Base Language: Scheme->C

The prototype Scheme^n implementation has used a port of Scheme->C [6] developed at Digital Equipment Corporation's Western Research Laboratory. It conforms to the Revised^3 Report on the Algorithmic Language Scheme [63], with some extensions such as additional I/O procedures and C data and procedure interface routines.

Scheme->C incorporates a Scheme-to-C compiler which can generate standalone programs containing the core Scheme language library code with user defined source code. The use of C as the intermediate language simplified the porting process while still producing fully compiled executables (as opposed to byte-compiled systems).

5.3 Messages

In a distributed memory system messages are used to transfer data and control between processors (see Chapter 2). In Scheme^n, the distributed nature of the memory is not directly evident to the user but messages are used internally to provide the communication necessary for the added Scheme^n parallel features (see Chapter 3).

The foremost use of messages involves the initiation of remote tasks, generated through **remote** procedure calls, and the subsequent return of results. These messages transfer Scheme^n data from one processor's memory to another. Scheme^n data is also transferred via messages when lazy or shared variables are accessed by a remote process (see Chapter 4). Section 5.3.1 gives details of how Scheme^n data objects are transmitted.

Messages are also required to control the scheduling and allocation of remote tasks. The workload and load estimates associated with each process (see Chapter 4) are needed by neighbouring processes to balance the load of remote task allocations (see Chapter 4). Remote tasks may also be disregarded from evaluation by a remote process because of an already high workload. Such rejected remote tasks may be returned to the parent task unevaluated for redetermination of an appropriate allocation.

Remote tasks can be explicitly terminated by sending a dedicated terminate message to the evaluating remote process. As remote tasks are not explicitly allocated by the parent task by default, the final destination of the distributed remote is required from the remote process in order for a terminate message to be sent. This destination is returned to the parent process as a separate message.

To handle errors generated in a distributed system, Scheme^n also returns messages, encapsulating the error condition of a remote task, to the parent task. Further distributed control is also required to initialise the Scheme^n system to determine processor numbers and topology, and also to close down the Scheme^n processes properly.

To accommodate these different messages, which can be received by processors in a non-deterministic order, a fixed format message protocol is used. Every message sent between processors consists of a fixed length header followed by the message-specific data. The header contains the following fields:

- Type
- Identifier
- Source
- Destination
- System
- Size

The message type indicates which of the above message categories is involved. The identifier provides processor-unique identification, generated by the parent process for association with subsequent replies. Typically, the identifier matches the process-unique identifier of the future data structure used by the parent to represent the remote task result or to identify a lazy or shared data value (see Chapter 4).

The source field is the system unique identifier associated with the message's source process. The destination field indicates the process identifier for the final destination of the message. If source and destination processes are not direct neighbours, messages are routed through intermediate processes. At Scheme^n system initialisation, each process determines the shortest path to every other process in the system. The process identifier of the neighbouring process to which a message should be sent to reach each process by the shortest path is recorded. Any messages received that are not destined for that process are routed to the destination process. If the final destination is a neighbour of the intermediate process, the message is sent directly to the final destination, otherwise it is sent to a neighbour that is nearer to the final destination.

Where the destination is undetermined, such as when distributing a remote task for evaluation, the destination field has the value of -1. Processes may retransmit any message with no specific destination to any other process, or it may choose to handle the message itself.

The system field of the message header is intended for passing some Scheme^n specific information. An implementation can use this to send the latest local load estimates between communicating processes, reducing the number of explicit load messages required. The size field specifies the size, in bytes, of the remaining message.

5.3.1 Transmitting Scheme^n Data

When sending Scheme^n data from one processor to another, the data must be translated into a state that can be transmitted. Typically, message passing is performed by sending a block of memory of a specified size and reading this into an equivalent block of memory at the destination.

Due to Scheme's frequent use of dynamically resizeable list data structures to represent both data and code, and the weak type declarations used, many Scheme^n objects are represented internally using pointers [1]. This leads to Scheme^n data being scattered throughout memory rather than being assigned to a contigu-

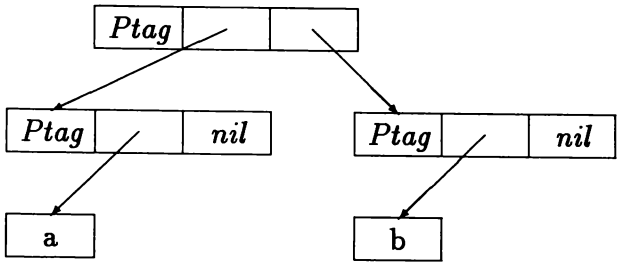


Figure 5.1: Scheme^n list representation of ((a) b)

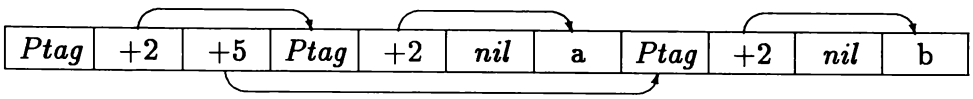


Figure 5.2: Contiguous representation of ((a) b)

ous memory section. For example, Figure 5.1 shows how the list ((a) b) (in dotted pair notation this is ((a . nil) . (b . nil))) is represented using five distinct memory elements; elements for a and b and three pair elements with a *Ptag* identifier and fields for the car and cdr of the pair. In the final cdr field of the lists, *nil* is used to represent the empty list.

To rearrange Scheme^n data into contiguous memory requires the data to be copied from its current locations, following any pointers encountered, to a sufficiently large block of memory. During the copying phase, the data can be converted into a form for transmission. In particular, absolute memory pointers must be removed.

The use of the built-in Scheme procedures to represent data structures as human-readable strings has been suggested as a simple but effective method of producing a transferable representation [40]. The method relies on run-time interpretation of the string produced to reconstruct the data structure at the destination process. Run-time interpretation (i.e. the **eval** function) is not supported in the Scheme standard.

Alternatively, the internal structure of the data structure can be retained, when copied to the contiguous memory block, with minor modifications to reduce its size. Absolute pointers can be replaced with relative offset values that are independent of the position of the object in memory. Figure 5.2 shows the same Scheme^n list as Figure 5.1 arranged in contiguous memory. The pointers have been replaced with offsets to the relevant data components.

List data can be compressed using *cdr-coding* techniques [5, 15]. A representation can also use the sequential order of the copied data and the unambiguous interpretation of a prefix notation to completely replace the pointers with the objects

Ptag	*Ptag*	a	*nil*	*Ptag*	b	*nil*

Figure 5.3: Prefix representation of ((a) b)

Ptag	3	*Ptag*	2	a	*nil*	b	*nil*

Figure 5.4: Prefix representation with list length of ((a) b)

they reference, regardless of size (see Figure 5.3).

Reductions in the size of the data sent can also be made by taking advantage of the static nature of the copied data. By recording the length of the list (immediately after the first *Ptag* of each list) the remaining pair tags can be removed (see Figure 5.4). For sufficiently large lists (length > 3) the size of the message is reduced and the explicit information about list length can be useful in regenerating the list on the destination process. Scheme^{n} uses this representation to format lists to be sent.

Other Scheme^{n} data types are also converted to a compact representation for sending. Some data types, such as integers, can not be easily compressed and are copied verbatim, while other data types, such as characters, can be reduced in size due to the byte sized representation possible in the message format. The following data types require further explanation of the message format used:

Procedure Procedure data types, which contain a reference to the executable code segment used to evaluate the procedure, are formatted by replacing the absolute memory pointer with an address relative to some fixed location of the executable (such as the address of the startup procedure). When the relative offset is passed to a process with an identical executable image, the correct code address can be reconstituted by adding the absolute address of the equivalent fixed address on the destination process.

This method of passing procedures requires exact copies of executables to be used for all communicating processes. A more complicated method could be used in situations where the process executables are dissimilar. Such methods would involve some memory, processor, and executable independent identification that uniquely identifies a procedure. Note that not all procedures are given explicit identification by the programmer so procedure names do not suffice. Lambda expressions, for example, are not associated with any unique identification, but it is desirable to have remote access to such procedures. Assuming that dissimilar executables are constructed from combinations of

the same compiled object files (separately compiled sources are assumed to be distinct procedures), the compiler could assign object file unique identifiers that will be identical in each executable.

Closures When evaluating a procedure the environment in which the evaluation takes place is defined by the procedure's closure. The closure defines the scope of variable names [1]. It specifies the set of all variables, including procedures, which are defined externally to the procedure and can be referenced. A closure is implicitly associated with all procedural objects and cannot be explicitly manipulated. In Scheme, the closure of a procedure is fixed by the context in which the procedure is defined making it *lexically* or *statically scoped* [63]. Alternatively, a language can be dynamically scoped with the environment of a procedure evaluation defined from the context of the procedure call. Lexical scoping has the advantage that all references to non-local variables can be determined at compile time and does not change with each call.

When a procedure is passed to a remote process, either as the procedure to be evaluated or as an argument, the environment in which the procedure is to be evaluated, as defined by its closure, must also be transferred. As the closure of a procedure includes all variables that can be referenced, the full closure can be large in size. It is seldom that any procedure will require access to all accessible variables and the passing of all details of this structure can be wasteful. For this reason, Scheme^{n} always passes the closure of a procedure using the lazy future construction (see Chapter 4).

For procedures which do not access externally defined variables, the closure need not be referenced and little overhead accrues from the presence of the lazy future. When a closure variable is accessed remotely, Scheme^{n} sends a request message for all variables within that specific closure. For nested closures, one level of closure is retrieved at a time, up to the level where the accessed closure variable resides. As the closure now resides on the remote process, any changes to closure variables by the remote procedure will not affect the source process. If changes by the remote process should be reflected in the source process, as is the case with sequential processing, then the relevant closure variables must be defined as shared variables (see Chapter 3).

For procedures and variables which are globally defined and exist in the top-level closure, which includes all built-in Scheme procedures, a closure data

structure is not normally defined as these objects are visible by all procedures. Top-level variables in Scheme^n are only global to a single process and each process can have a different value associated with each. In some uses this is acceptable if the top-level environment is processor-dependent and relates to properties of individual processes such as the process identifier and process workloads. For static values such as global procedure definitions, the presence of multiple exact copies on each process serves the required purpose. To produce the same behaviour as sequential evaluation where changes made by a process to a top-level object are immediately observed in any other access, either all accesses to a top-level variable should be made by a single process or shared variables are required.

Continuations The passing of continuations between processes has not been included in the prototype Scheme^n implementation and the semantics of such passing have not been fully explored. The following paragraph describes how continuations might be introduced into Scheme^n.

It is assumed that continuations are never passed between processes. References to them can be made but these are always implicitly replaced with lazy futures. Remote references to the lazy future do not cause the continuation to be retrieved or evaluated on the remote process as with standard lazy processing. Instead, the current remote task is terminated and control is returned to the source process where the parent task will initiate the appropriate continuation.

Futures The only types of future passed between processors are the lazy and shared future. Other future types are converted to lazy futures for transmission (see Chapter 4). A reference to the passed lazy or shared future refers back to the source process to retrieve a value substitute.

5.3.2 Maintaining Structure Sharing

The implemented Scheme^n message passing format has the side effect that any data sharing in a data structure is destroyed. Data sharing can be an important feature of some data structures whose behaviour relies on `eqv?` equality or on destructive operations such as `set-car!` and `set-cdr!`. In other situations, data sharing is only used as an implementation efficiency of memory and computation

and can be removed with no adverse effects. Extra care must be taken by the programmer when the existence of shared data is required for correct evaluation, for example data shared structures can not be given as the first argument of an **append** procedure call.

Circular, or self-referential, data structures are a special case of data sharing. Circular structures can only be created using destructive operations. They require special handling by procedures such as print where normal list traversal leads to an infinite loop.

Because data sharing can be a necessary requirement of some algorithm implementation, a method is needed to allow such structures to be passed from one Scheme memory space to another as part of the normal remote evaluation handling. The previous method for passing data must be augmented to ensure the copy procedure preserves the data sharing attributes of an arbitrary scheme data structure.

The following methods could be used to pass data shared structures including circular structures.

1. As data is copied, replace or mark the data in some manner. If the data is referenced again, the presence of marked data indicates that the data has already been processed and the copy procedure will need to reference the processed data rather than reprocessing it and thus generating an unwanted replica of the shared data. When the entire structure has been processed, marked data must be unmarked.

   Copying in this manner requires some means of marking the data. This could be accomplished using the garbage collection mechanism. However, using a mechanism such as this, because it uses a processor-global status, requires synchronisation with all other light-weight processes which complicates the message passing mechanism unnecessarily, as well as degrading performance.

2. To avoid modifying the data, a list of all references made can be maintained during the copying process. This list could then be searched before each new object is copied and if a reference already exists then an index to the object should be used instead. To avoid explicit construction of this list, the stack can be used. However, searching such a list linearly requires $O(n^2)$ operations.

3. To improve the search performance of method 2, a hash table can be used to store pointers to the already copied objects.

The data sharing capabilities of the above methods work only for those structures that are passed in a single operation. For example, arguments of a single remote call will maintain data sharing but arguments of different remote calls will contain distinct copies of any shared data.

The scope of the data sharing can be extended to include structures involved in several operations but this requires the knowledge of previous references to be maintained for the lifetime of the operations. For method 1, this means the necessary synchronisation with other light-weight processes will need to be extended, interrupting other processing further. For methods 2 and 3, the list or hash table must be preserved. This will mean a larger list or hash table will be required and longer delays in searching the database.

Shared data can be taken to its logical limit where all operations between Scheme^n memory spaces will maintain data sharing by recording all references. Shared data on this scale is equivalent to emulating shared memory on a distributed memory system (see Chapter 2). When taken to this extreme additional issues must be addressed. These include allowing garbage collection of externally referenced objects to occur, which involves the updating of object addresses (in a table hashed on addresses this will mean a rehash of the entire table is necessary) and the removal of objects no longer in use and also the effect of destructive operations on shared data. When a copy of an object is passed to another memory space, any destructive modifications to that object will need to be recognised so that a new copy can be passed, either as a replacement for the existing copy, or when the object is to be passed again.

5.4 Communication Requirements

For each Scheme^n process there are several operations that require prompt servicing above and beyond the needs of a standard Scheme process. These operations include:

- The sending and receiving of messages from other Scheme^n processes.
- The processing of requests for remote task evaluation and the subsequent processing of the evaluated replies.
- The ability for each process to execute several Scheme evaluations at once.

Because of the synchronous nature of communication between Transputer processes, a process which writes an inter-process messages cannot proceed until the receiving process is ready to read the message. When the message is not expected by the receiving process or the frequency of communication is variable, it is difficult for a single process to adequately check for the appearance of incoming messages while performing other computations. The situation is made more difficult when messages can be received from many different sources in a non-deterministic order. Delays in reading messages have a cumulative effect as the writing process is also delayed which causes other processes wishing to communicate with either process to be likewise delayed.

To avoid these delays the message handling operations can be performed asynchronously to the main computation by forking light-weight processes (LWP). A LWP shares the processor and memory of a Scheme^n process. Each LWP is given access to the CPU at regular intervals so that each LWP can proceed with its computation. Each LWP has its own evaluation stack but all LWPs have access to a common shared memory pool. The LWPs can communicate with each other using shared global variables with correct synchronisation being maintained by the use of semaphores. If the number and order of inter-process messages that are to be sent, received and processed by each process is unknown, using LWPs that can act independently of other LWPs allows several tasks to continue uninterrupted.

The Scheme^n system includes LWPs specifically designed to handle the input and output of messages (see Figure 5.5). Each process communication channel is assigned an associated input server LWP and an output server LWP whose sole job is to read or write messages for the single channel respectively, queueing the messages for further processing. While there are no incoming or outgoing messages these I/O server LWPs are suspended and do not effect the processing of other tasks. When process I/O is required, these servers can process the communication with a minimum of delay. Two additional LWPs are also included in each process to handle remote requests and remote replies independently to other computations that are proceeding (see Section 5.6 and Section 5.9).

Even with these dedicated LWPs present, there are times when a remote task cannot proceed because it requires information from other active remote tasks. While this LWP is suspended, the Scheme^n process is not being properly utilised. To avoid this, the Scheme^n system has been designed to allow more than one remote task to be evaluated at once by each Scheme^n process. This is accomplished by

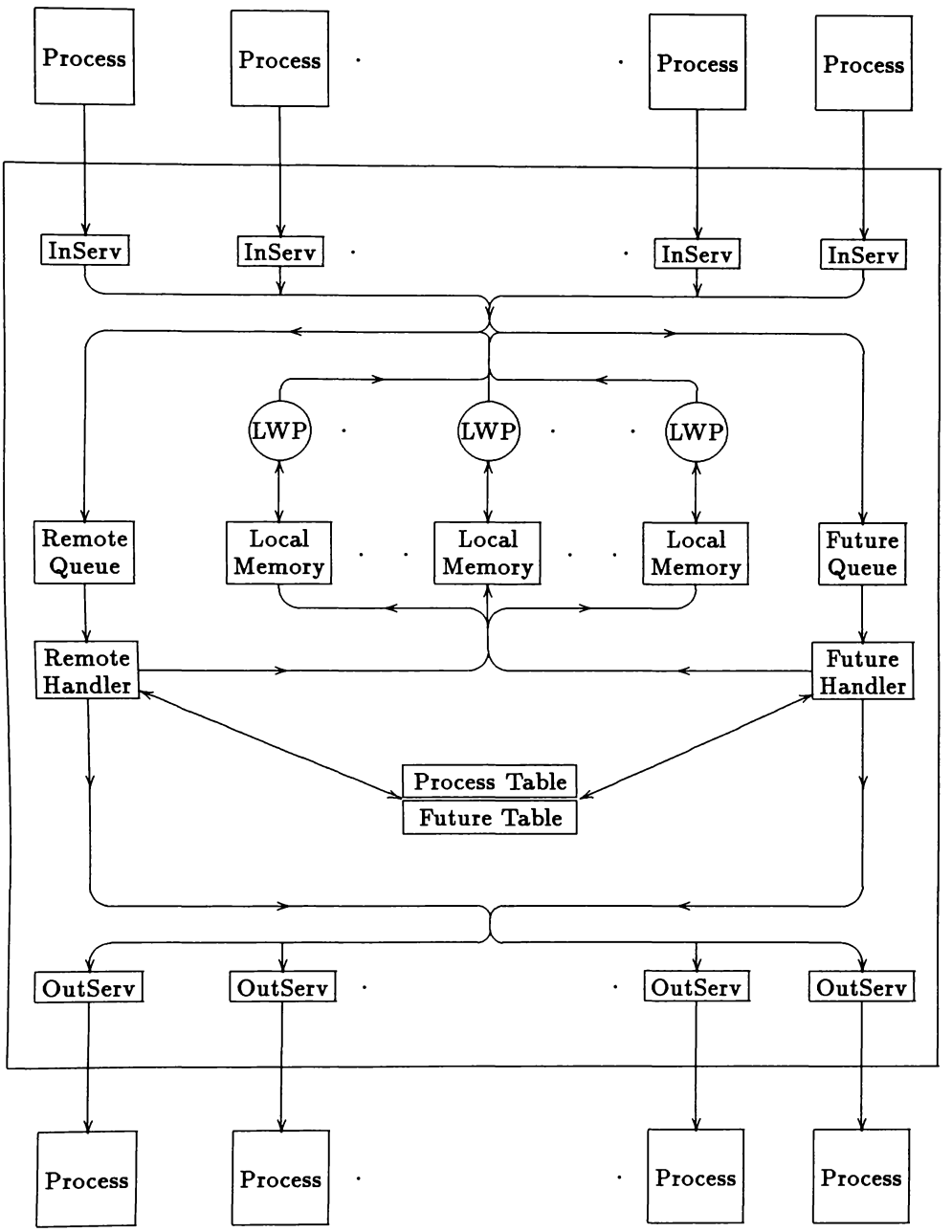


Figure 5.5: LWP configuration

forking LWPs for each remote task that is initiated.

5.5 Remote Initiation

The introduction of parallelism into a Scheme procedure is accomplished with the **remote** procedure call as discussed in Chapter 3. When the **remote** predicate argument evaluates to a non-false value, indicating the procedure should be evaluated remotely, the procedure initiates the following sequence of events:

1. Creates a remote message.
2. Adds the remote message to the remote handler input queue (see Section 5.6).
3. Creates a future object.
4. Returns the future as the initial result of the **remote** call.

A remote message (see Section 5.3) is initially represented as a Scheme^n list. If the remote message is to be sent to other processes, the Scheme^n representation of the remote message, which cannot be directly passed between process memory spaces, must be converted into an equivalent message structure that can be transferred between processes (see Section 5.3.1). The decision to convert the remote message into an externally readable form is delayed until it is necessary, avoiding the overhead of converting to and from the external representation when a task is to be evaluated locally.

A future data structure is also created, to be returned to the initiating task, as the immediate result of the **remote** call. The future created has the same identifier as the associated message and an undetermined type. An entry is also made in the future table (see Chapter 4), using the future id as a key, so that the future data structure can be found from its id. Having received an undetermined future object, the initiating task can then continue in the evaluation of other functions while the **remote** call is being processed. When the result of the **remote** call has been evaluated the value is returned to the process using the future handler LWP by replacing the undetermined future with a determined future having the result as its value (see Section 5.9). In all further processing, the exact value will be used in place of the future.

5.6 The Remote Handler

The remote handler LWP is used to process the remote requests initiated by a `remote` procedure call. The remote handler continuously performs the following steps:

1. Reads a remote message from its input queue.
2. Calls the load balancing function to determine whether the remote should be evaluated locally or if it should be sent to a less loaded processor.
3. If the remote is to be evaluated by another process, the remote message is queued for transmission.
4. Otherwise, if the remote is to be evaluated locally, a LWP is forked to evaluate the task.
5. Reads the next remote message.

Remote requests are received from both local LWPs and from LWPs of other processes via the input servers. The remote handler determines whether the remote request should be satisfied by the current process or whether another process can more effectively evaluate the task based on current local and external workload estimates. The load balancing procedure indicates either that the local process should evaluate the remote task or it specifies a channel direction in which to send the remote request or, if no process is currently available, the remote request is rejected and returned to its source.

If the remote task is to be evaluated by another process and the remote request was generated locally then the request is first converted from its Scheme list representation into an equivalent message structure. Remote requests are then placed on the queue of the output server responsible for the outgoing communications of the channel specified by the load balancer.

For remote requests that are to be satisfied by the local process, the remote handler forks a LWP to evaluate the remote task. The newly forked LWP is then responsible for the task's evaluation and the returning of its result to the source process. Where necessary a remote destination message is sent, by the remote handler, to the remote request's source process to inform it of the final destination of the remote task. This is done so that if it becomes necessary to terminate the remote task the source process can send the kill message to the correct process.

When load balancing, there may be no processes available to evaluate another remote task or the remote handler may fail to identify a process that is currently idle. Failure to assign a process is an indication that the multiprocessor system is fully loaded. Attempting to assign the remote task could lead to an overload of available computing resources that may cause the entire evaluation to fail. A remote request that cannot be successfully assigned to a process is returned to its source process as a rejected remote. Further processing of rejected remotes is carried out by the source process as it has the easiest access to information concerning the requirements and necessities of evaluating the remote task. Because the associated future structure on the source process retained a copy of the remote procedure and arguments, the remote reject message does not need to return this information reducing the size of the returned message.

5.7 Remote Evaluation

If it has been determined that a remote task should be evaluated by the current process the remote handler forks a LWP to evaluate the task. The remote task consists of a Scheme procedure and optional Scheme arguments. A remote evaluation involves the following operations:

1. Define a default exit continuation.
2. Apply the Scheme procedure to its arguments.
3. Return the result of the evaluation to the parent LWP.

To allow a Scheme procedure to be terminated prematurely a continuation is constructed. The continuation is assigned to the Scheme **exit** procedure. In situations where the Scheme procedure invokes an error, the **exit** procedure is called. For a remote task, the **exit** procedure will not exit the Scheme^n system but will instead return control to the exit continuation. An error message can then be returned as the "result" of the remote task, propagating the error from a child task to its parent. Operations by the parent task on an error value will cause a further error to be generated which will be returned to its parent. Errors will therefore be returned to the top-level task with child tasks being allowed to terminate cleanly. The procedure of a remote task may also explicitly call the **exit** procedure to terminate the remote task returning a value to its parent. A remote task can also

redefine the **exit** procedure so that clean-up operations can be performed. The exit continuation is used to allow a parent task to terminate a child task. When a remote task is killed the **exit** procedure is called allowing the remote task to perform clean-up operations and to return a partial result to its parent.

When a result has been produced the value is returned as the determined value of the remote task's associated future. If the remote task was not evaluated on the source process of the remote call then the result must be converted to an equivalent message so it can be transferred back to the parent task. The remote reply message is then queued to the output server handling the communication channel in the direction of the remote task's source process. The message retains the message id of the remote task so it can be matched with the correct future on the source process. The message is routed through the necessary input and output servers along a shortest path between the remote tasks source and destination. At the remote task's source the message is queued on the future handler queue for further processing. For remote tasks which were evaluated by the task's source process the result can be directly added to the future handler queue without the need for message conversion or routing through the I/O servers.

5.8 Input and Output Servers

For each message channel, separate LWPs are forked to process the messages received and those to be sent. Because the read and write operations for a message channel are synchronous, a channel server is designed to read and handle each message quickly so that the communicating process is not delayed. Each server should operate asynchronously to other local LWPs to avoid unnecessary delays. In particular, server dependence on Scheme data should be avoided as access to Scheme memory can be delayed because of the need for synchronisation with other Scheme LWPs and the randomness of the garbage collection instances. By using privately allocated memory, queues for LWP communication and other LWPs to perform the tasks required, the server can process messages with high throughput.

5.8.1 Output Server

When a message is to be sent, it is queued by a LWP to the appropriate output server queue. For each process in the system, the process table specifies a message channel number which is part of a shortest path from the current process

to the destination. This table is initialised at system startup. The correct output server queue is found from the process table. By sending a message via the specified message channel, the message will be routed, by each intermediate process, along a shortest path until the message reaches its destination.

The following steps are performed by each output server:

1. Wait for a message to be written to the message queue associated with the specified channel and remove it.
2. Assign the current value of the system load to the system field of the message header.
3. Write the message to the channel.
4. Wait for next message.

5.8.2 Input Server

The following steps are performed by each input server:

1. Wait for a message to be written by some other process to a given message channel.
2. Read the fixed-length header and an optional variable-length message body (see Section 5.3).
3. If the destination of the message has been specified and it does not match the current process identifier then queue the message to the correct output server queue.
4. For messages which are destined for the current process or whose destination has not been specified, the input server processes each message depending on the message type.
5. Wait for next message.

The following actions are performed based on the message type:

Message Type	Action
Remote Requests	place message on the Remote Handler queue
Remote Replies	place message on the Future Handler queue
Kill	place message on the Future Handler queue
Load	update relevant loads
Destination	place message on the Future Handler queue
Close down	broadcast message and terminate process

5.9 The Future Handler

The efficiency of the system will be maximised if LWPs are not required to wait unnecessarily. Also, starting the execution of new tasks will be hampered if suspended LWPs are present as these will be holding computing resources that may be required by the new tasks. The returning of results should therefore have priority over initiating new tasks as existing tasks may be able to complete on receiving a remote result. Also the returned results can affect whether or not the tasks being initiated are actually required and the quicker this can be determined the less unnecessary work is performed.

A separate LWP is used to handle the messages received from LWPs and other processes that contain the evaluated results of remote tasks. By separating the operations of remote initiation from remote resolution, results can be processed independently from the processing of new remote tasks with no delays caused by intermediary remote requests.

The future handler repeatedly processes the following operations:

1. Extract incoming message from the future handler queue.

2. Convert the remote result from a message structure into an equivalent Scheme data structure, if necessary.

3. Replace matching future data structure with the determined value.

The future handler receives messages, via a queue, from the LWPs associated with remote tasks. These messages have been routed from the remote task's destination process back to the task's source process by the input and output servers of processes in a shortest path from destination to source. The input server on the source process will recognise that the destination of the remote reply message has been reached and will queue the message onto the future handler queue. For remote tasks which have been evaluated on the source process, the remote reply is added directly to the future handler queue.

The message representation of a Scheme data object is converted into the equivalent Scheme structure (see Section 5.3.1). Results from remote tasks are returned to the parent task via the associated undetermined future, created when the remote call was initiated. The future is found using the future table by matching the identifier of the message and future table entries added when remote requests are initiated. The matching table entry, which can be removed once a remote reply is received, gives the memory location of the future data structure.

The future, which should currently be undetermined, is replaced with a determined future type with the returned result as its value. If the future has been terminated and has a terminated type then the result has been returned during the time between the sending of the kill message by the source and the actioning of the kill by the destination. Any result returned by a terminated remote task may still be required as the exit procedure of the just completed remote task can not be performed as a result of the kill message and no termination value will be returned. If no value is returned to the source process any task requiring a value will be permanently suspended which will probably cause the computation to never complete.

Any LWPs waiting on a future to be determined are reactivated. Reactivated LWPs check the type of the future, or futures, on which they were suspended. If the future is now determined they can proceed with their own evaluation. For LWPs waiting on futures which are still undetermined the LWP is again suspended to be awakened when another remote reply is received. If a result is returned before the parent task requires it the parent can immediately access the returned value without being suspended or delayed by the remote task evaluation.

Chapter 6

Analysis

6.1 Introduction

To demonstrate the behaviour of the language implementation described in Chapters 3, 4 and 5, a number of examples are given. Each example uses a small Scheme^n procedure to illustrate a particular attribute of the implementation. A more complete application is studied in Chapter 7.

The behaviour of the system is studied for a variety of logical process configurations. The physical topology of the available hardware was arranged in a ring configuration (see Figure 6.1). The Helios operating system allows users to specify a logical configuration which can be automatically or manually mapped onto the physical topology as described in Chapter 5. The logical configurations have been chosen to be representative of a range of configurations that are commonly used. The topologies used for analysis are:

- fully connected (see Figure 6.2)
- ring (see Figure 6.3)
- binary tree (see Figure 6.4)
- star (see Figure 6.5)
- grid (see Figure 6.6)

For comparison purposes, the number of processors used in each topology are roughly equal. The fully connected, ring and star topologies each have eight processes. For the binary tree, seven processors are required to fully specify a tree

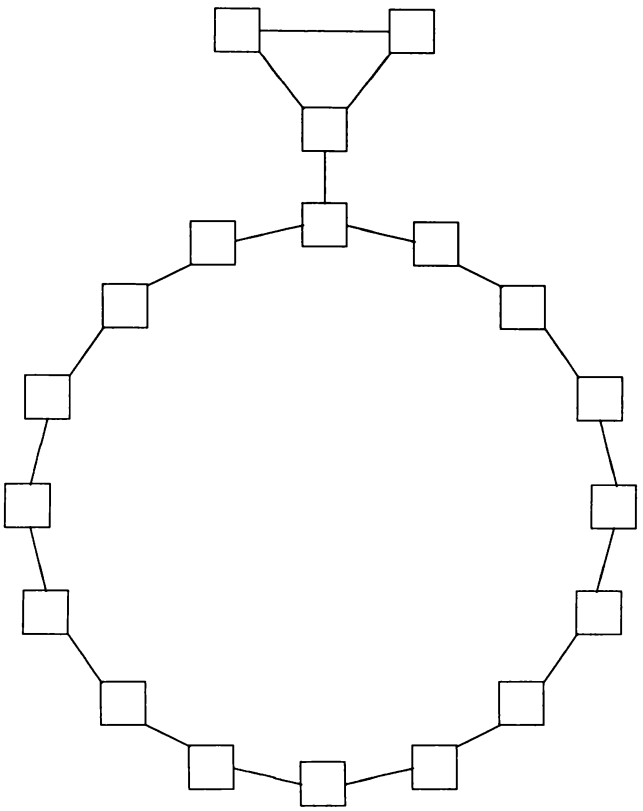


Figure 6.1: The Physical Transputer Configuration

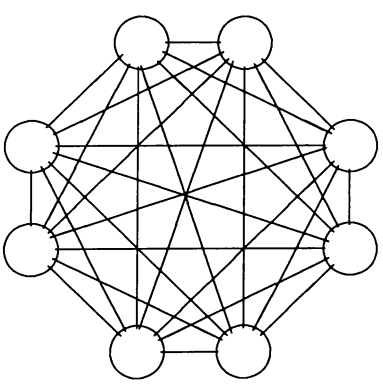


Figure 6.2: Fully Connected Topology

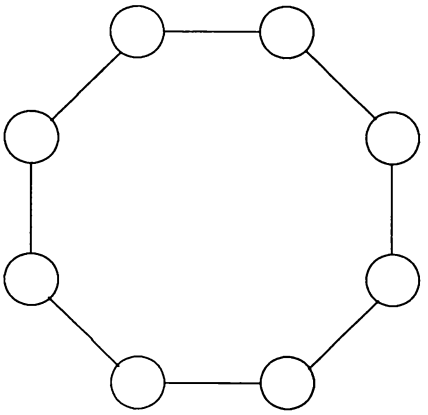


Figure 6.3: Ring Topology

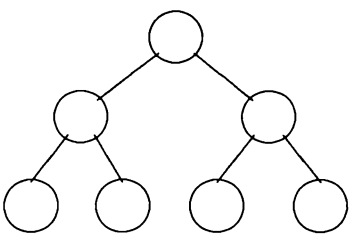


Figure 6.4: Binary Tree Topology

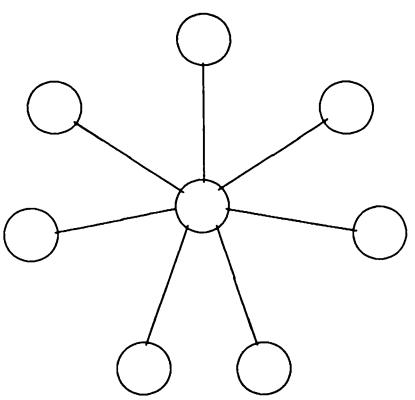


Figure 6.5: Star Topology

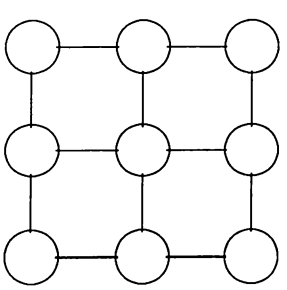


Figure 6.6: Grid Topology

of depth two. For the grid topology, nine processors are used for a three by three two-dimensional grid. All communication channels are bidirectional.

Although nineteen processors were available, the analysis has been limited to roughly eight processors. Eight processors were deemed enough to clearly illustrate the general behaviour of the system. A restriction to eight processors was also necessary in the fully connected topology because the large number of communication channels and associated I/O server light-weight processes required could not be easily accommodated in the memory available to each processor and still retain sufficient memory for user processing.

Particular emphasis is given to the dynamic process load of the system under different conditions. The ability of the system to evenly distribute the concurrent tasks of the application amongst the available processors is central to its performance characteristics. Load balancing is performed in a distributed fashion by each processor based on relayed information of neighbouring processor loads. A processor's view of the current system load is therefore only an estimate of the true system load with the possibility of each processor having very different viewpoints. The accuracy and import of these estimates is examined and the resulting behaviour is graphed.

6.2 Remote Overhead

The overhead introduced when evaluating procedures concurrently is important to the user when determining the granularity of tasks that can be effectively evaluated concurrently. The time required to evaluate a task should be significant with respect to the remote overhead in order for concurrent evaluation to appreciably reduce the overall evaluation time. The extra time added to perform the evaluation in parallel must be smaller than the smallest concurrent task for concurrency to be considered. Figure 6.7 illustrates the effect of overhead on the concurrent evaluation of two tasks.

For our implementation of Scheme^n, we can quantify the minimum remote overhead or *latency* to give us an indication of the minimum beneficial task size. The overhead for a remote call is dependent on several factors. The number and configuration of processes, the physical hardware and topology and other activity in the system will all have an affect on the behaviour of a remote call. The decision as to where a remote call is evaluated is particularly important to the overhead incurred.

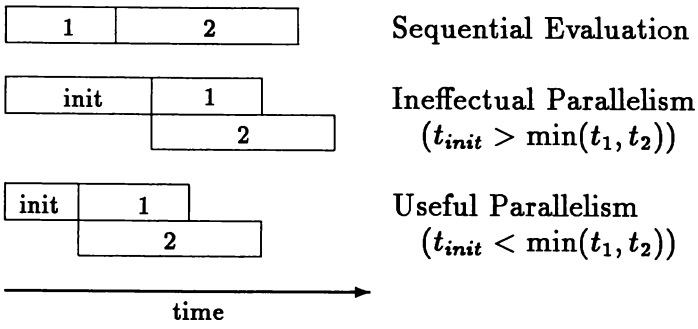


Figure 6.7: The Effect of Overhead

The time spent load balancing, transmitting the remote request and the remote reply and updating process load estimates can change significantly with changes in the static and dynamic processing environment.

Using a simple system of only two processes situated on physically adjacent processors the variability of the remote call's behaviour can be controlled. With only two possible places for a remote call to be evaluated, on the source process or on the neighbouring process, the load balancing decision is well defined with load estimates reflecting closely the true load of each process. The effect of inter-process communication is also minimised using physically adjacent processors and the smallest messages possible. For a remote call, the smallest request involves a procedure with no arguments, and the smallest result is a Scheme constant such as `#T`. The overhead determined from these conditions will provide a lower bound for the overhead in all other situations.

To isolate the overhead of the remote call from other computation, a loop containing a remote call to a simple procedure will be compared to a similar loop which evaluates the same procedure without the remote call. The difference in times between the two loops will be entirely due to the remote call. The Scheme^{n} procedures used for the measurements are given in Figure 6.8.

Each of these procedures are compiled, `null` is the procedure used in the remote call, **`basetime`** is used to measure both the loop and `null` procedure time and **`overhead`** adds the remote call to the **`basetime`** loop. Each of the remote calls will be satisfied on the neighbouring process since its load will always be less than the local process. Touching the result of the remote procedure call causes the source process to suspend itself until the result of the remote call has been returned. Without the touch, the loop would continue with the next iteration, before the remote has necessarily completed, invalidating the measurements.

Using the code of Figure 6.8, 100,000 iterations of **`basetime`** were found

```
(define (NULL) #T)

(define (BASETIME n)
  (do    ((i n  (- i 1)))
         ((zero? i) #T)
         (null)
))

(define (OVERHEAD n)
  (do    ((i n  (- i 1)))
         ((zero? i) #T)
         (touch (remote #T null))
))
```

Figure 6.8: Remote Overhead Measurement Code

to take 690 milliseconds, or 0.0069 milliseconds per iteration. For 10,000 iterations of **overhead** the time is 113,000 milliseconds or 11.30 milliseconds per iteration. From the difference between these two measurements, the overhead of a remote call is 11.30 milliseconds to two decimal places.

Compared to a basic Scheme **cons** operation which takes 1 microsecond, and the **apply** procedure which takes 35 microseconds, the overhead of a remote call is significant. Although the overhead is not excessive, it warrants a medium to coarse parallel task granularity. Fine grain parallelism would exhibit poor performance on this system, as the overhead introduced extends the computation time rather than reducing it.

6.3 Communication Overhead

From Section 6.2, the total overhead of a remote call with no arguments and a result of **#T** is 11.30 milliseconds. Some of this overhead is involved in communicating the relevant information between processes. Communication involves the tasks of formatting the data, transmitting the message and reforming the data into its original structure (see Chapter 5). For the simple configuration of two processes used in Section 6.2, each remote call required four messages to be sent between the processes. The two main messages are the remote request and the remote reply.

The remaining two messages sent are load messages, as described in Chapter 4, which are sent to neighbouring processes each time the local load changes. The number of load messages generated by a remote call is dependent on the con-

figuration of the processes and on other message traffic. Each time the local load of a process changes a load message may be sent to all its neighbouring processes. For the given situation a maximum of four load messages could have been generated for each remote call; one each for when the source process waits for the remote reply and is awakened on reception and one each for the neighbouring process as the remote evaluation is started and completed. For the current situation only two load messages were sent, when the source process is woken and when the neighbouring process begins evaluation of the remote. For the other two cases when load messages are possible, other messages are being sent between these processes and, since these messages will transmit the current load of the process, no specific load message is required. Of course, some processing time is still needed to determine when such load messages are not required.

The communication overhead for each message sent will depend on the communication hardware available, the size and type of messages to be sent and on the destination. From [54] the time to transmit a message using the Helios Posix read/write routines can be calculated from a program provided in its appendix. For the available hardware (see Chapter 5), the communication performance can be stated as

$$t_{total} = 925.262420 + 0.565932N \ (r = 0.999982)$$

where t_{total} is the transmission time in microseconds and N is the message size in bytes.

Using this transmission formula the fraction of the time for a minimal remote procedure call that is attributable to the communication performance of the hardware can be calculated. The smallest remote request message consists of a message header of 24 bytes and a list of length one containing the message representation of the *null* procedure, with no arguments, which requires a further 10 bytes giving a total message size of 34 bytes. On the available hardware, this remote request message will take 944 microseconds to transmit. The remote reply message consisting of a `#T` value and header is 25 bytes in length and takes 940 microseconds to transmit.

Each load message consists of only the message header of 24 bytes and will take 939 microseconds to transmit but some of this time is done in parallel with other processing. The use of asynchronous I/O in the Scheme^{n} design and the Transputer's ability to perform I/O in parallel with CPU activity, means the time to transmit each load message does not affect performance unless there is contention

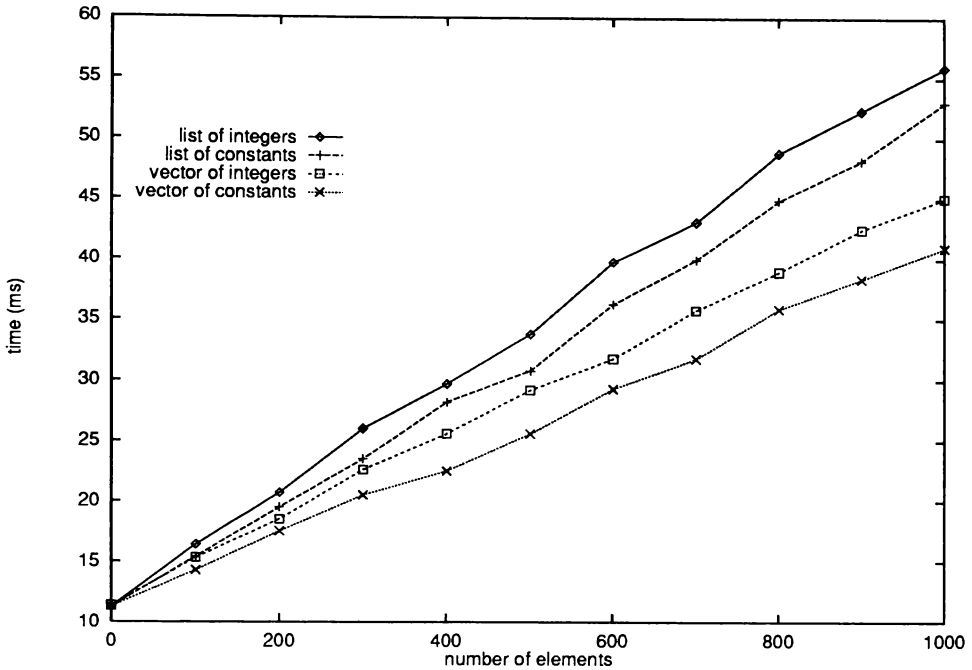


Figure 6.9: Communication Performance - I

with other communications. The message initialisation time of 925 microseconds is therefore the only contribution by each of the two load messages to the time for the remote evaluation. The total transmission time therefore accounts for approximately 3.7 milliseconds of the 11.3 millisecond remote procedure call.

The remaining 7.6 milliseconds of the remote evaluation time is taken up in creating the remote and its equivalent message representation, in initialising the LWP on the destination process to evaluate the remote procedure and in processing the result. It is difficult to further divide the remaining time for each of these operations because of the relatively small time intervals involved compared to the measurable interval of 10 milliseconds and the overlapping of these operations with each other and the already accounted for message transmissions.

Where the arguments to a procedure or the results of the procedure call are larger than those considered, the size of the equivalent message increases, increasing the transmission time and the time to convert the Scheme^n data to and from the message representation. The time to construct a message from Scheme^n data or convert such a message back to a Scheme^n data structure is dependent both on the size of the data involved and its type. Figure 6.9 shows the communication performance between adjacent processors for both lists and vectors of differing lengths and element types. Figure 6.10 gives similar results for lists of multi-precision integers.

Figure 6.9 shows a clear linear dependence between the communication

times and the number of Scheme elements transmitted. From these experimental times, the following basic formula for communication time can be calculated using linear regression. The times are expressed in milliseconds with N representing the number of elements in the list or vector.

list of integers	$11.836 + 0.04502N$	$(r = 0.99914)$
list of constants	$11.145 + 0.04155N$	$(r = 0.99946)$
vector of integers	$11.918 + 0.03376N$	$(r = 0.99845)$
vector of constants	$11.214 + 0.02995N$	$(r = 0.99910)$

The size of the messages for lists and vectors are nearly identical so the transmission times are the same but from the graph it is apparent that the formatting of lists for message passing is more computationally expensive than a vector of the same length. The difference is due to the time required to traverse a list and allocate the memory for each list node.

Similarly, formatting integers is more time consuming than formatting constants such as `#T`, `#F` and the empty list. Each integer requires 4 bytes as compared to 1 byte for each Scheme constant, which requires additional transmission time as well as time to construct the message. For example, the time to transmit a list of 1000 integers takes approximately 1.7 milliseconds longer than 1000 Scheme constants which accounts for approximately half the observable difference in communication performance.

As the size of a message increases, the contribution of transmission time to the remote procedure call becomes less significant. The communication time becomes dominated by the message formatting operation. There are two orders of magnitude difference in the time to transmit a message compared to the preparation of the message for transmission.

The different time scale of Figures 6.9 and 6.10 is due to the extra information contained in a multi-precision integer data structure when compared to integers or Scheme$^{\text{n}}$ constants. A multi-precision integer data structure contains a vector of integers, a sign field and a length field which need to be formatted for transmission. Figure 6.10 shows the communication performance for lists of multi-precision integers of roughly 15 and 30 decimal digits which can be represented with 2 and 4 integer terms respectively. The larger size of each multi-precision integer data structure also produces a discrepancy in the linear trend observed in Figure 6.10 between the number of elements and the communication time. At approximately 220 and 380 elements respectively, the 2 and 4-integer component graphs exhibit a sudden

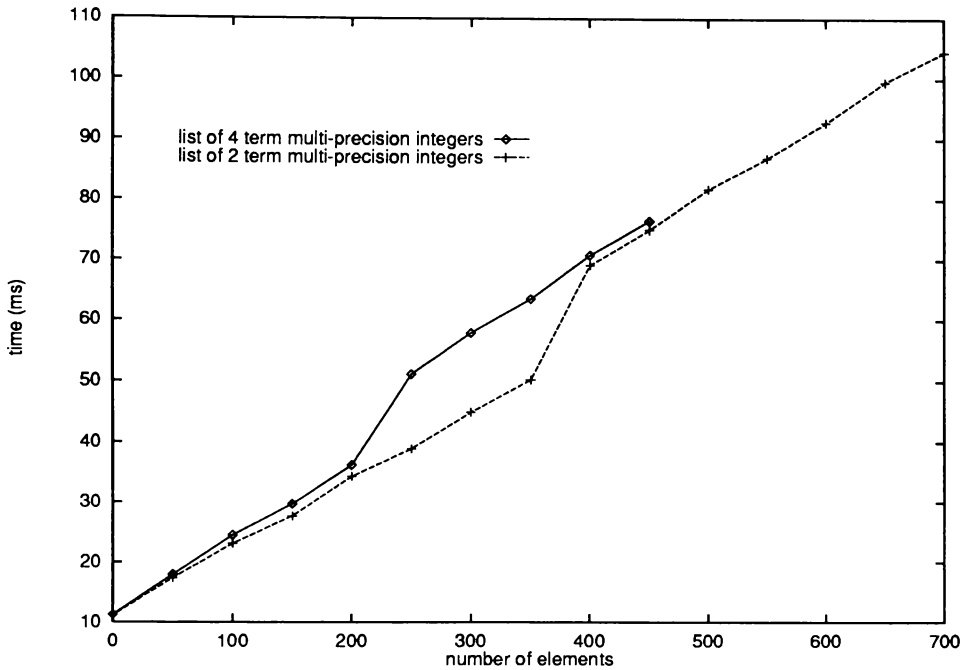


Figure 6.10: Communication Performance - II

jump in their communication times before continuing their linear behaviour. At these points the size of the message being constructed reaches approximately 5000 bytes in size. The Scheme^{n} implementation initially assigns 5000 bytes of memory for each message sent. For messages larger than this, 10,000 bytes are assigned and the data is re-converted. This increases the linear constant in the overhead of larger messages but the rate of increase of the communication time with an increase in the number of elements is relatively unaffected. The implementation limit of 10,000 bytes for a single message also accounts for the limit in the size of the multi-precision integer lists sent. Using linear regression on the experimental communication times for multi-precision integer messages above and below the 5000 message boundary gives the following formula, expressed in milliseconds, where N is the number of list elements.

list of 4 component integers	$11.640 + 0.1230N$	($r = 0.99917$) for N =0–200
	$19.370 + 0.1274N$	($r = 0.99928$) for N =250–450
list of 2 component integers	$11.608 + 0.1106N$	($r = 0.99955$) for N =0–350
	$21.793 + 0.1187N$	($r = 0.99948$) for N =400–700

Similar communication performance occurs with vectors of multi-precision integers. Data structures containing a mixture of data types will exhibit communication times related to those given but dependent on the ratio of each data type.

6.4 Implicit Future Checking

The introduction of parallel computation to the Scheme environment can also influence the performance of purely sequential computation. With implicit checks performed to test for the presence of undetermined futures, additional code is added to all strict operations increasing the evaluation time even when there is no parallel activity. As the future checks affect the performance of all parallel and sequential code, small increases in the time to evaluate each operation can be significant. If the performance penalty for implicit future checking is high the benefits of any concurrent evaluation have to be larger in order for the increased expenditure of resources to be worthwhile.

In the Scheme^{n} implementation, attempts have been made to minimise the effect of implicit future checking on the performance of sequential code in particular. This has been done at a cost in program size with future checking being performed in-line.

To illustrate the effect of implicit future checking on performance, a simple sequential loop containing several common procedures, a stylised sequential binary recursion procedure, a more complicated parallel primality testing example and several parallel integer factorisations (see Chapter 7) will be used. The time to evaluate each of these procedures will be measured with and without implicit future checking and also with and without type checking. The results of each evaluation are given in Table 6.1.

The results of Table 6.1 can only be used as a rough guide to the effects of both type and future checking because of the wide variation in times but some comments can be made. From these results and personal experience, the effect of implicit future checking on evaluation times is roughly ten percent with type checking disabled and five percent with type checking enabled. For procedures which are type checked, future checking need only be performed when type checking fails as a valid type indicates no future is present. This is evident from the table results where smaller increases in evaluation time when type checking is enabled. For numeric operations, the effect of type and future checking is minimal as in-line checks are always needed, independent of future checking, to distinguish integer and floating point numbers.

Reasons for some changes in the evaluation times given in Table 6.1 are not immediately obvious. For the recursion example, which only performs proce-

	Type Off		Type On	
	Future Off	Future On[a]	Future Off[b]	Future On[c]
loop	318	329 (3.5%)	334 (5.0%)	334 (5.0%/ 0.0%)
recursion	1768	1935 (9.4%)	1768 (0.0%)	1935 (9.4%/ 9.4%)
prime test[d]	412	418 (1.5%)	412 (0.0%)	419 (1.7%/ 1.7%)
factorise[e]	3428	3696 (7.8%)	3608 (5.2%)	3734 (8.9%/ 3.5%)
factorise[f]	249	275 (10.4%)	268 (7.6%)	280 (12.4%/ 4.5%)
factorise[g]	918	1001 (9.0%)	977 (6.4%)	1014 (10.5%/ 3.8%)

[a](increase compared to Type Off, Future Off)
[b](increase compared to Type Off, Future Off)
[c](increase compared to Type Off, Future Off/ increase compared to Type On, Future Off)
[d]$N = 263130836933693530167218012159999999$
[e]$N = 765041185860961084291 \cdot 811$
[f]$N = 537913 \cdot 100357 \cdot 661$
[g]$N = 541150847 \cdot 18251$

Table 6.1: Effect of Implicit Future Checking

dure calls and integer arithmetic, the evaluation time should not be affected by future checking since procedure calls are not strict operations and future checking of integers should impose no significant overhead. Possible reasons for the relatively significant increases of 9.4% may be due to differences introduced by the Helios C compiler or assembler. The increase in code size may influence how branching in the assembler code is performed. Other examples, such as the prime test of a 36-digit prime in Table 6.1, are less affected by future checks than expected.

6.5 Load Balancing

The performance of the Scheme$^{\text{n}}$ implementation is highly dependent on the behaviour and performance of the dynamic load balancing mechanism. For overall efficiency all processors should be kept busy, with tasks evaluated as close as possible to their parents to reduce communication overhead.

The behaviour of the load balancing mechanism is dependent on the number and size of tasks that require assignment. Many small parallel tasks mean process loads are continuously changing and the load estimates used by each process to balance the load can become out-of-date quickly. Inconsistencies in process loads increase the likelihood of a misinformed choice for the best destination process which can unnecessarily load an already busy process when other processes are idle.

Many more tasks than processes can also lead to situations where the system becomes overloaded and tasks must be rejected and returned to their parent

for evaluation when resources become available or the evaluation becomes critical. The search for a likely host for a remote evaluation can be futile when the system is highly loaded and it can be more profitable to reject the task, returning it to the parent process, and wait for some significant reduction in process loads before attempting to assign the task. Excessive load balancing activity as processes are continually passed from loaded process to loaded process will only use valuable CPU time that can be better utilised in reducing the system load rather than increasing it. Applications whose design does not fit within the available system capabilities can exhibit poor performance as task contention overrides the application's own evaluation. Redesigning the application to better match resources and requirements, by combining together small tasks, can achieve better processor utilisation.

To maintain up-to-date process loads of neighbouring processes messages must be sent whenever load changes occur. The frequency of load messages must be chosen so as to keep neighbouring processes adequately well-informed as to the current load status of the process without significantly increasing the system overhead.

Load messages are only sent to neighbouring processes. Changes to the load induced by load messages sent by other processes are not propagated. This reduces the message traffic while keeping the nearby processes, which are the most likely to send tasks, up-to-date. As a process's estimate of system load is garnered from local processes only, the true system load can be quite different, especially when processes are separated by many intervening processes. The effect of this locality can cause ineffectual load balancing as local load peaks hide lightly loaded processes from view. Processes with few neighbours can become isolated if their neighbours are loaded but do not themselves produce any remote tasks. Other processes will not attempt to pass tasks to these processes because of their load and hence processes attached to loaded processes can be overlooked.

When task creation is not evenly performed by all tasks, the communication capability of the processes can become a major load balancing criterion. Manual assignment of tasks may be required if the placement of tasks has a significant effect on performance. It is advantageous to assign tasks which produce the most remote tasks to processes which have the most neighbours and therefore the best chance of distributing the task load evenly. Because of the dynamic nature of task creation, information about individual task fertility is unknown to the load balancer and explicit destinations should be given by the user in these situations.

```
(define (remote-recurse n)
  (if (> n 0)
      (touch (remote #T remote-recurse (- n 1)))
))
```

Figure 6.11: Code to demonstrate load balancing

To illustrate the behaviour of the load balancing mechanism, the simple recursion procedure, **remote-recurse** listed in Figure 6.11 will be analysed. The example consists of a straightforward recursion where each instance is performed as a remote evaluation. The example does not exhibit any useful parallelism, as each instance of the recursion is immediately dependent on the result of the remote task initiated so only one instance is active at any given time, but it does provide a succinct example of load balancing on a loaded system. The controlled linear sequence of remote calls and remote task assignments produced is easier to describe than a more complex system where the order of the remote calls is no longer serial.

Figures 6.12 and 6.13 show the sequence in which remote tasks are assigned to processes of an eight-process fully connected configuration when evaluating **(remote-recurse 32)** for two separate runs. Each figure depicts two representations of the sequence of remote calls. The line graph indicates the order in which remote tasks are assigned to the available processes. For example, the sequence depicted in Figure 6.12 begins with the evaluation of **(remote-recurse 32)** on process 0 which generates a remote call. The remote call is load balanced and the associated procedure **(remote-recurse 31)** is assigned, by the load balancing mechanism, to process 2, as indicated with a line from process 0 to process 2 in the figure. Similarly, **(remote-recurse 31)**, begins a remote call which is sent to process 7. This sequence of remote calls continues until **(remote-recurse 0)** is started, as the thirty-second and last remote task, on process 4. The other graph shows, for the same sequence, the overall pattern of process communication for the **remote-recurse** evaluation. A line is drawn from each source process, which generates a remote task, to the destination process, where the task is evaluated.

From these graphs, a rough pattern of task assignment can be seen as the load is distributed over the available processes and each process is given a set of tasks to evaluate. Of the possible paths in the fully-connected system, many different paths are followed as the process loads dynamically adjust themselves. The exact behaviour, however, is irregular and dissimilar as it is dependent on the values of

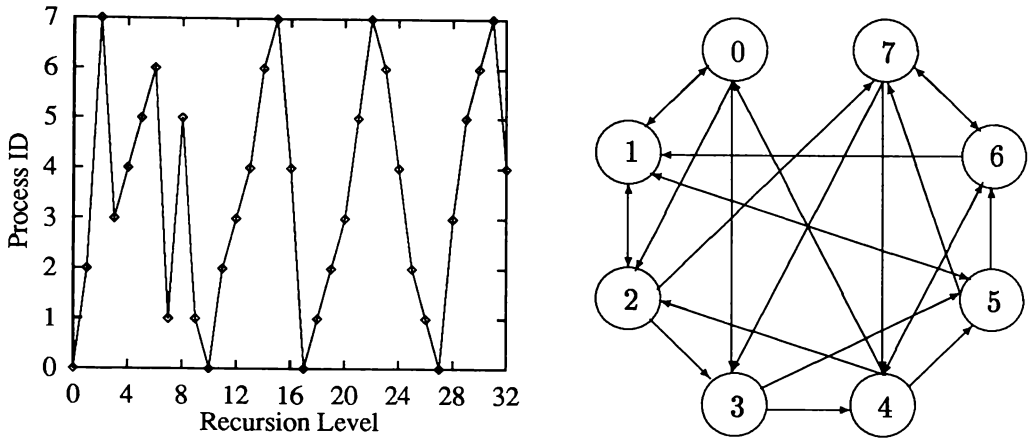


Figure 6.12: Load Balancing Sequence I

the system load estimates before the procedure is started and on past and present process activity and loading. Even when the initial load estimates are identical the behaviour of the system can vary as the exact order in which messages arrive from different sources can change with each evaluation, affecting load estimates and hence the load balance.

Tables 6.2 and 6.3 show the load estimates for Figures 6.12 and 6.13 respectively. Each table gives load estimates for the processes before and after the **remote-recurse** example is evaluated. A system load of 1000 represents a system where each process has one active LWP. An inactive LWP only contributes a load of 500 to the local load of a process. Two different system load estimates are given; the *current system estimate* and the *transmitted system estimate*. The transmitted estimate is the last estimate of each process that was distributed to that process's neighbours. The system load of a process is only distributed when LWPs on that process start, finish, or are suspended, changing its local process load. The current system load estimate is the process's most up-to-date estimate, calculated from its local process load and from other process's transmitted system estimates as they are received.

For this example, where the system is fully connected, every process will receive the same set of load messages. For partially connected configurations, only load estimates of neighbouring processes are sent to a process, although all such neighbours will receive the same estimates whenever a process's local load changes.

The system estimates after each example is run are generally higher than the system load before. The level of system activity during the evaluation is reflected in the artificially high estimates when all processing has completed except for the

Process ID	System Load Estimate			
	Current		Transmitted	
	Before	After	Before	After
0	257	872	241	896
1	59	644	54	945
2	53	671	30	700
3	59	659	52	807
4	59	655	52	849
5	59	647	53	891
6	59	642	53	908
7	59	660	54	749

Table 6.2: Load Estimates I

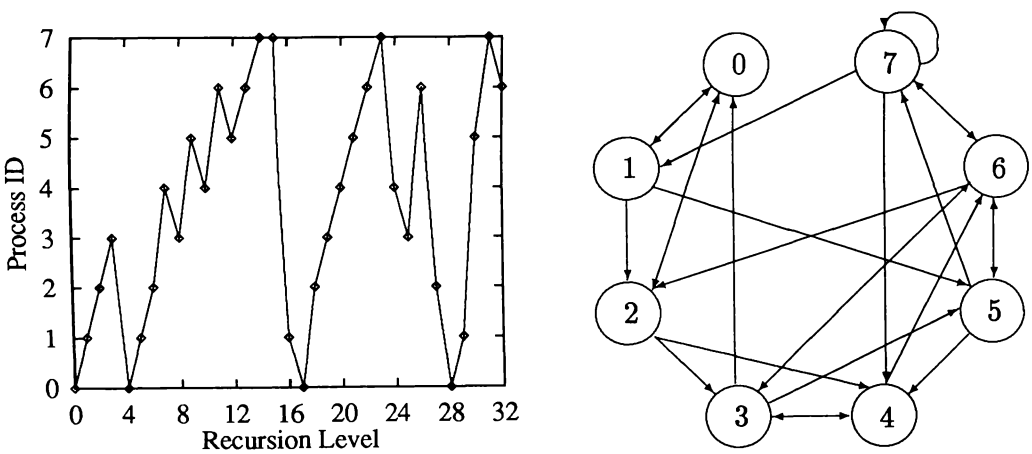


Figure 6.13: Load Balancing Sequence II

Process ID	System Load Estimate			
	Current		Transmitted	
	Before	After	Before	After
0	846	1001	846	1026
1	643	804	673	799
2	637	800	721	820
3	632	799	772	848
4	627	778	816	1034
5	623	768	853	1100
6	621	761	871	1160
7	616	751	910	1253

Table 6.3: Load Estimates II

main Scheme^n read-eval-print loop. The estimates are calculated from system loads which infer a load from a process that has since completed. The load estimates will more closely reflect the true system load as further remote tasks and communications are circulated through the system. In the second example of Figure 6.13, which has artificially high system load estimates before the evaluation begins, the first few remote task assignments are not evenly distributed as the load incorrectly indicates activity on idle processes. As the loads change, and remote tasks are forced onto seemingly loaded processes, more up-to-date estimates are forthcoming and the remaining remotes are more evenly assigned to processes.

The transmitted load figures tend to lag behind the current load estimates. If the load on a system is increasing, transmitted loads will be less than current loads and similarly, a decreasing system load tends to produce higher transmitted estimates than the relevant current estimates. This behaviour is due to changes in the local load of neighbouring processes affecting the current system load estimate of a process but, as load is not propagated, this change does not generate further load messages. The processes which complete first will therefore transmit, as their last load message, the highest system loads due to the activity of the other processes.

From the table for the second example, process 7 has the highest transmitted load after the evaluation as it is the last process to be utilised (at `(remote--recurse 14)`) and therefore the first to complete its recursion. Similarly, process 1's transmitted and local loads are nearly the same as it is the last to finish processing. The load of process 0 reflects not only the load from the remote-recursion example but also the load of the main Scheme^n read-eval-print.

Another effect of the way system load is measured, is that the system load estimate of a process with a changing local process load is the least accurate. This is due to the system load of a process's neighbours being partially due to a process's own process load. When that load changes, the process's neighbour's system estimates are changed to reflect this, but these changes are not propagated back to the source of the changes. Of course, the load change also does not affect the system estimates of non-neighbouring processes but as these processes generally do not have every neighbour of the load-changing process as a neighbour the effects of the load change are less significant. As an example of this, for the second evaluation of the recursion example given, process 7 has the most accurate current system estimate after evaluation while process 1 has the least accurate. This is the reverse of the accuracy in transmitted load estimates.

The load balancing mechanism also exhibits another characteristic not evident from the graphs and tables provided. At points where every process has been assigned the same number of tasks, the load balancing mechanism has some difficulty correctly assigning the next remote task. Having a process evaluate more than one task at a time is not desirable and only occurs when no other idle process is available. As the transmitted system estimates lag behind the current system estimates, a process in a currently evenly-loaded system which receives a remote task can mistakenly believe that other processes are less loaded, and rather than evaluate the remote task itself, will send the task to the least loaded process based on those estimates. In sending the task the system load of the receiving process is updated and will more closely reveal the even loading of the processes. Based on these more up-to-date estimates, the receiving process may then decide to evaluate the remote task or send the task to another process depending on the relevant load values. Eventually, the system load of a process will reveal the true system load and the remote task will be evaluated.

For the recursion example given, as the number of remote tasks reaches 7, 15, 23 and 31, every process has equal numbers of LWPs. At these points, roughly four attempts at assigning the eighth, sixteenth, twenty-fourth and thirty-second remote task are made. After four attempts the system load estimates were close enough to the true system load as to choose the current process, even though that process is already loaded. For other tasks in the example, the initial assignment was correct.

For other process configurations, where all processes do not have direct connections to all others, the load balancing characteristics are different. With communication distances between processes greater than one, the system load estimates are more localised as is the distribution of remote tasks. Even process utilisation may not occur when non-neighbouring processes are not reached because the neighbouring processes between have high loads. Figure 6.14 illustrates the load balancing behaviour for the `remote-recurse` example on a tree configuration. The process connections in Figure 6.14 represent the available connections for the tree configuration and not the pattern of communication as displayed in Figures 6.12 and 6.13. As the configuration used is no longer symmetrical, the process ID information is important. The process IDs have been assigned such that the line graph can approximate the configuration structure. To further illustrate the behaviour for the tree configuration, Table 6.4 gives the connection matrix for the communication

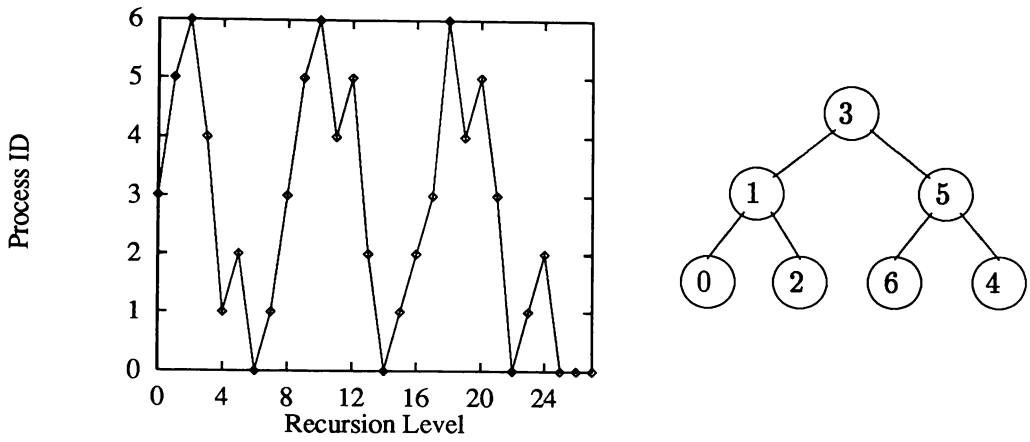


Figure 6.14: Balancing for tree configuration

paths used in the evaluation of `(remote-recurse 28)`. Each element of the directed connection matrix shows the number of communications from the row-process to the column-process. The undirected connection matrix shows the same information but the direction of the communication is not taken into account. From these matrices, it is clear most communication occurs within each sub-tree of the root process, process 3. The top-left and bottom-right sub-matrices of Table 6.4, which represent the left and right sub-trees of Figure 6.14 respectively, involve significantly more communications than the other sub-matrices which represent cross-communication. When communication is required between the sub-trees it normally occurs as control is returned to the root process.

Most communications involve neighbouring processes. Very few remote tasks have needed to be redirected from their initially process. The line graph for this example shows only two communications of distance three and three communications of distance two were required.

The load balance is not exactly even. Processes 4 and 6 were assigned only three remote tasks while process 0 was assigned six. This unevenness occurs as the process load on each process increases and process 0, which is a leaf process of the left sub-tree, is unaware of the lighter loads on the leaf processes, 4 and 6, of the right sub-tree because of the distance and intervening process loads.

Directed

Process ID	0	1	2	3	4	5	6
0	-	3	0	0	0	0	0
1	0	-	3	1	0	0	0
2	3	0	-	1	0	0	0
3	1	0	0	-	0	2	1
4	0	1	0	0	-	2	0
5	0	0	1	1	0	-	2
6	0	0	0	0	3	0	-

Process ID

Undirected

Process ID	0	1	2	3	4	5	6
0	-	3	3	1	0	0	0
1	3	-	3	1	1	0	0
2	3	3	-	1	0	1	0
3	1	1	1	-	0	3	1
4	0	1	0	1	-	2	3
5	0	0	1	3	2	-	2
6	0	0	0	0	3	2	-

Table 6.4: Connection Matrix

Chapter 7

Application: Factorisation

7.1 Introduction

To demonstrate the capabilities and performance of the Scheme^n language, a parallel implementation of a large integer factorisation package has been implemented. This implementation is not intended to present any newly developed parallel algorithms for factorisation but rather to demonstrate the features of our language of interest, Scheme^n, by examining the process of implementing a well-researched, but computationally difficult, application and analysing its performance.

The main feature of Scheme^n demonstrated by the factorisation package is its ability to perform speculative computation. Numerous factorisation algorithms exist [39, 65], many of which are straightforward to implement, but whose performance in the factorisation of a randomly chosen integer is difficult to gauge. Upper bounds of the execution time of some algorithms are available but the actual time required for the factorisation of a particular integer can differ radically from these upper limits. With additional information about the form of the integer and the nature of its factors the performance of each factorisation method can be better analysed. For a general factorisation package such individual analysis can not always be done and the use of several competing algorithms in parallel can achieve good results for a selection of integers.

This chapter first introduces the factorisation algorithms implemented, specifying any inherent parallelism found. Additional algorithms to determine primality, necessary for the factorisation package, are also described. A description follows of the complete parallel factorisation package and how the useful parallelism of the algorithms have been combined and implemented in Scheme^n. Analysis of

the run-time behaviour and performance of the package is given.

7.2 Integer Factorisation Methods

Numerous methods have been developed for finding the unique factorisation of an integer into a product of prime numbers [39, 65]. The ability of each method to find a factor of a given integer, N, is obviously dependent on the magnitude of N, but it is also dependent on the size and number of its prime factors and on properties such as the form of $N-1$ or $N+1$. How each method is influenced by such number theoretic properties affects which method will find a factor first, implying no one method is the 'best' in all situations. For example, the simplest method of trial division (see Section 7.2.1) will quickly find small factors, while methods such as Fermat's method (see Section 7.2.4) are more suited to finding factors of comparable size. It is this fact that makes the concurrent use of several different methods, for the determination of prime factors of a given integer, appealing.

This section concentrates on describing the methods chosen for implementation. The methods include trial division (see Section 7.2.1), Pollard's $\rho-1$ method (see Section 7.2.2), and Morrison and Brillhart's continued fraction method (see Section 7.2.3). These methods where chosen to represent a cross-section of methods that, between them, show good performance for most integers. The trial division method is best suited to factorising integers with small factors. Pollard's $\rho-1$ method's success is dependent on the structure of $\rho-1$, where ρ is a prime factor, with the best performance occurring when $\rho-1$ has many small factors. Morrison and Brillhart's continued fraction method shows good performance for integers that are not readily factored by the previous methods. Section 7.2.4 describes some of the other methods that could also be introduced to the factorisation package implemented.

7.2.1 Trial Division

Trial division is the most obvious method for finding factors of an integer N. By attempting to divide N by all primes between 1 and $\sqrt{N}$ the prime factors of N will be determined. Implementing the above method has a drawback; the primes between 1 and $\sqrt{N}$ must first be generated before the division can proceed. For small primes, a table can be constructed and stored which can then be used for each instance of factorisation. To provide a table of all primes up to $\sqrt{N}$ would

however require considerable storage space even for values of N of approximately 30 digits. The number of primes less than $\sqrt{10^{30}}$ is approximately 10^{13} [65, page 376]. It may however be appropriate to store a partial table of primes up to some more manageable limit say 10^3 which only contains 168 primes.

Although it is only necessary to trial divide N by primes it is permissible to include composite numbers as well. If N is divided by all integers less than $\sqrt{N}$ starting from 2, the first integer p which divides N will be a prime factor of N. Continuing the process with N/p will find further prime factors of N.

Dividing N by all integers between 1 and $\sqrt{N}$ includes many unnecessary trial divisions. The number of unnecessary divisions can be reduced by removing as many composite numbers as possible. By removing all multiples of the very small primes a large reduction in unnecessary divisions is possible. For example, excluding all multiples of 2, 3 and 5, removes 22 out of every 30 possible trial divisions from consideration. Having removed these multiples, 2,3,5 and the integers $30k \pm 1$, $30k \pm 7$, $30k \pm 11$, $30k \pm 13$ are still contenders for division. The number of trial divisions left is still much larger than the number of primes that must be tried but the determination of which of the remaining trial divisors are unnecessary becomes increasingly difficult to determine. To retain efficiency in the trial division process it is preferable to attempt the remaining unnecessary trial divisions rather than determine their redundancy.

When determining all prime factors of N and a prime factor p has been found, the trial division method can proceed to factor N/p from the trial divisor it is currently using as N/p will have no factors less than p if N had no factors less than p. Because a factor can be present in N to some power other than one, the factorisation of N by p should be repeatedly tested. The upper limit of trial divisors is reduced from $\sqrt{N}$ to $\sqrt{N/p}$ so if $p > \sqrt{N/p}$ the algorithm can terminate with the value N/p guaranteed to be a prime factor of N.

Possible Parallelism: The trial division method can be easily converted to parallel operation by dividing the range of possible divisors, between 1 and $\sqrt{N}$, and using different processors for each partition. The range can be evenly divided amongst the processor based on the interval size, or each partition can be chosen to have approximately equal numbers of primes. The second partitioning scheme concentrates the processing on the small numbers, improving the chance of factorisation.

When a factor is found, the parallel evaluations can be terminated and the

new smaller range, 1 to $\sqrt{N/p}$, can be divided among the processors. A record of the last divisor trialled in each range should be maintained to avoid repetition.

A consequence of the parallel search for factors, is it is no longer possible to automatically guarantee that factors found are prime. Each factor found should be checked for primality, and if necessary factored further.

7.2.2 Pollard's $\rho - 1$ Method

This method, published in [56], is based on the following theorem,

Theorem 7.1 (Fermat) See [38, page 39]. *If p is a prime, and $a \not\equiv 0 \pmod{p}$, then $a^{p-1} \equiv 1 \pmod{p}$ or equivalently $p \mid a^{p-1} - 1$.*

Let p be a prime factor of N and $a \not\equiv 0 \pmod{p}$ then from Fermat's Theorem $p \mid a^{p-1} - 1$. If $p - 1 \mid Q$ then as $a^u - 1 \mid a^{uv} - 1$ we have $a^{p-1} - 1 \mid a^Q - 1$. Combining this with Fermat's Theorem we have $p \mid a^Q - 1$. Since $p \mid N$, p can be found by applying Euclid's algorithm to $a^Q - 1 \pmod{N}$ and N.

Pollard's method requires suitable values of Q be chosen such that $p - 1$ is likely to be a factor of it. The most sensible choices for Q are those which have many small prime factors. The method relies on the possibility that although p may be large, $p - 1$ may factor into many small prime factors. In this situation, $p - 1$ will divide Q relatively quickly and potentially large factors can be discovered with ease.

By constructing a series of values Q_i and periodically checking $(a^{Q_i} - 1 \bmod N, N)$ for a non-trivial common divisor a factor of N can be found. Rather than constructing Q from the product of all small integers, which would include an over abundance of the small primes, a product of primes with limited duplication of factors is preferred. By generating a sequence of integers, beginning at 2, with multiples of small primes removed, provides a practical method of fashioning possible values for Q that does not consume unnecessary time in determining an exact set of primes. This sequence of multiplicands can be generated in the same manner as the trial divisors of the trial division method of Section 7.2.1.

Since $Q_{i+1} = Q_i p_i$, where p_i is the i^{th} integer in the generated sequence of multiplicands and $Q_1 = 1$, we have

$$(a^{Q_{i+1}} - 1) \equiv ((a^{Q_i} \bmod N)^{p_i} - 1) \pmod{N}. \tag{7.1}$$

Rather than storing Q_i, which can be large in magnitude, the value $a^{Q_i} \bmod N$ is stored instead.

Using a specialised binary exponentiation method that uses modulo arithmetic, the number of multiplication steps, all involving integers $\leq N$, required to generate each successive $a^{Q_{i+1}} - 1$ value is $(\lceil \log_2(p_i) \rceil + \sigma(p_i))$ where $\sigma(p_i)$ is the number of ones in the binary representation of p_i [39, page 443]. A similar number of division steps are also needed to reduce the numbers at each step modulo N.

It is not necessary to apply Euclid's algorithm for every Q_i in the sequence. Applying Euclid's algorithm at regular intervals to the current $(a^{Q_i} - 1) \bmod N$ and N is sufficient to finding a factor of N[1].

The number of iterations required by the algorithm is independent of the choice of a. The algorithm will succeed as soon as $p - 1 \mid Q$ for some prime factor p of N. It is possible for the above method to fail in the rare cases when the largest factor of $p_i - 1$, for all factors p_i of N, are identical. Partial success is possible if several factors have a common largest factor for $p_i - 1$.

The method can also fail for particular choices of a when N has the form $(a^m - 1)/k$. Choosing a Q such that $m \mid Q$ will cause $a^Q - 1 \equiv 0 \pmod{N}$ and no factor of N will be discovered even when Q contains all factors of some $p - 1$. In the algorithm presented, the sequence of Q values used is constructed by forming a product with the previous instance of Q so that once $m \mid Q_i$ all subsequent Q_i values will also be divisible by m. To avoid this problem we must either choose a different a and restart the algorithm or select a Q such that $Q \not\equiv 0 \pmod{m}$. By removing the factor of Q which made Q a multiple of m will allow us to continue to use the Q so far constructed. However, if the factor discarded from Q is a factor of $p - 1$ then we will fail to find the corresponding factor p of N which may have been possible if a different value of a was instead chosen.

When finding the complete factorisation of N and a factor p is found, the factor must itself be checked for primality because of the possibility that multiple factors of N were discovered at the same time. The Pollard $\rho - 1$ method cannot be used to further factorise p but the value $a^{Q_i} - 1 \bmod N$ can be used to find any remaining factors of N/p. Since $A \bmod q = (A \bmod pq) \bmod q$ we can extract the value $a^{Q_i} - 1 \bmod (N/p)$ from $a^{Q_i} \bmod N$ and use this without having to regenerate

[1] Riesel [65] incorrectly suggests that an accumulated product of $a^{Q_i} - 1$ values should be formed. This step is unnecessary since each previous Q_i value is a factor of the latest Q_i and as such a factor p of N will be found if and only if $p - 1$ divides the latest Q_i.

the product Q_i so far produced.

Possible Parallelism: The method described above defines a recurrence relation for a sequence of Q_i values. This implicitly imposes a sequential order to their evaluation precluding any parallel activity. The Q_i values, however, need not be strictly limited to the elements in this sequence, although those values are the most likely to contain the factors of $p-1$, and other combinations can be attempted. It is difficult to see how distinct Q values can improve the method since it is the computation of a value $a^Q - 1 \bmod N$ with Q having many factors which will provide the best results. Calculating several Q_i values and evaluating $a^{Q_1 Q_2 \cdots Q_k} - 1 \bmod N$ is not computationally worthwhile since it is the evaluation of the exponential which requires the most processing time.

7.2.3 Morrison and Brillhart's Continued Fraction Method

This method [39, 49, 65] attempts to find numbers x and y that satisfy

$$x^2 \equiv y^2 \pmod{N}. \tag{7.2}$$

And assuming $x^2 \geq N$ then $\exists k \in \mathbb{Z}, 0 < k < N$ such that

$$\begin{aligned} x^2 &= kN + y^2 \\ x^2 - y^2 &= kN \\ (x+y)(x-y) &= kN \end{aligned} \tag{7.3}$$

where

$$0 \leq y < \sqrt{N} \leq x < N \tag{7.4}$$

and

$$x + y \neq N.$$

With these conditions, the greatest common divisor of $x - y$ and N is a non-trivial factor of N.

The method requires suitable choices of x be made that satisfy equation 7.2. If several values of x were chosen such that

$$x_i^2 \equiv a_i \pmod{N} \tag{7.5}$$

and each a_i could be factored over a set of m primes $p_1, p_2, \ldots, p_m$ such that

$$a_i = p_1^{\alpha_{i1}} p_2^{\alpha_{i2}} \ldots p_m^{\alpha_{im}}.$$

Then since

$$(\prod x_i)^2 \equiv (\prod a_i) \pmod{N},$$

we can use a product of these factored a_i to find a value of a that is a perfect square giving us a value for y in equation 7.2.

Choosing x at random in the range $\sqrt{N} \leq x < N$ would make this method no better than trial division. However, if the values of a_i were kept small then the likelihood that each a_i would factor over a given set of primes would be increased. Since, from equation 7.5,

$$x_i^2 - k_i N = a_i. \tag{7.6}$$

for some positive integer k_i, a_i will be small if x_i is a good approximation to $\sqrt{k_i N}$. The calculation of $\lceil \sqrt{k_i N} \rceil$ for different values of k_i can be used to generate sufficient a_i so that a perfect square can be formed. The magnitude of a_i for such choices of x_i is between zero and $2\sqrt{k_i N} + 1$ which is a smaller range than using random choices of x_i, provided $2\sqrt{k_i} < \sqrt{N}$ or equivalently $k_i < N/4$, but is still extensive enough to require many choices of k_i before a perfect square is likely to be formed.

Alternately, we can choose a particular value for k and find rational, rather than integer, approximations to $\sqrt{kN}$ using a continued fraction expansion. The *nth* convergent rational approximations of the continued fraction expansion of quadratic irrationals such as $\sqrt{kN}$ have a property akin to equation 7.6. For the *nth* convergent A_n/B_n of $\sqrt{kN}$ we have the identity

$$A_n^2 - kNB_n^2 = Q_n, \tag{7.7}$$

for some integer Q_n, $0 <\mid Q_n \mid < 2\sqrt{N}$. The range of values for Q_n is smaller than the range from finding integer approximations to $\sqrt{k_i N}$, for $k_i > 1$, so the chance of successful factorisation of Q_n is improved. The continued fraction method also has advantages over finding $\sqrt{k_i N}$ integer estimates in that the values A_n and Q_n for successive convergents of $\sqrt{kN}$ are calculated using a recurrence relation which is computationally more efficient than finding integer approximations to square roots of large numbers.

The choice of primes to include in the factor base to be used to factor Q_n

of equation 7.7 is dependent on the value of kN. The only odd primes, p_i, which can be factors of any Q_n of equation 7.7 are those in which $(kN)^{(p_i-1)/2} \bmod p_i \leq 1$ [39]. The value of k chosen should allow the factor base to contain the most small primes to increase the chance of factorisation.

The number of primes to include in the factor base is also an important consideration in producing an effective practical method. Too few primes in the factor base will mean the Q_n values found will seldom factor completely and many attempts at factorisation must be made. Too many primes in the factor base will have the opposite effect in that relatively large numbers of Q_n will factor but more factorisations will be required since they are less likely to combine to give a perfect square. In addition, the increased number of trial divisions needed for factorisation of Q_n values over larger factor bases can impede the number of attempts. No definitive value for the most beneficial choice of m is forthcoming and heuristic analysis is suggested. For values of N with approximately 40 digits, 100 to 200 primes in the factor base appears to be a satisfactory number.

One way of increasing the scope of the factor base without increasing the number of trial divisions is to factor the trial Q_n by all m primes in the factor base and look at the magnitude of the remaining factor of Q_n. If this factor is less than p_m^2, the largest factor in the factor base then the factor must be prime [65, page 200]

Another improvement that can be included in a computer implementation of the algorithm involves avoiding trial divisions for Q_n which are unlikely to factor over the factor base. If a Q_n has few small factors in the factor base and the remaining cofactor of Q_n is relatively large then it is unlikely that it will completely factored. In such case where success is unlikely, the factorisation can be aborted and the next Q_n tried. Again, the decision to abort the attempt at factorisation requires heuristic analysis [65, page 209].

Possible Parallelism: The steps in the algorithm which are the most time consuming, since they must be carried out often, involve the generation of A_n, Q_n pairs and the attempted factorisation of Q_n over the factor base. Because of the recurrence relations used to generate the A_n and Q_n values, Morrison and Brillhart's method is inherently sequential. Some overlap is however possible between the generation of the Q_n values and the associated factorisations. By generating sequences of Q_n values in advance, factorisation can be attempted in parallel. To balance the evaluation time of the two operations, the task of factorising can be divided

between several processes (see Figure 7.2). A proper balance is achieved when the time required to factor a set of Q_n values exactly matches the time to generate the next set.

When a Q_n value factors over the factor base, the generation-factorisation cycle should be suspended to determine if a perfect square has been discovered. Care must be taken that the cycle can be restarted again, without loss of any A-Q pairs, if a perfect square is not forthcoming. The reduction of A_n, Q_n pairs, when Q_n can be completely factored, into a perfect square occurs relatively infrequently so any performance gains here will have little affect on the overall evaluation time.

Other methods for parallel implementation of this method have been suggested by Williams and Wunderlich [76] and Riesel [65].

7.2.4 Other Methods

The methods described thus far, comprise the implemented algorithms for the Scheme$^{\text{n}}$ large integer factorisation package. Their integration into a complete factorisation package, together with primality testing and auxiliary testing for greatest common divisors and perfect powers, is described in Section 7.4.

There are several other methods available for factorisation that have not been implemented. The package, as presented, was deemed sufficient to illustrate the potential of the Scheme$^{\text{n}}$ language, and the inclusion of other methods is fully possible. The benefits of including each method should be weighed against their use of computing resources.

Fermat's method [65] uses Legendre's congruence to find possible factors of N. That is, finding x and y such that $N = x^2 - y^2 = (x+y)(x-y)$. The method chooses an x starting at $m = \lceil\sqrt{N}\rceil$ and determines if $m^2 - N$ is a perfect square, if not try $m+1$, and so on. As $(m+1)^2 = m^2 + 2m + 1$ it is not necessary at each iteration to square m, adding $2m+1$ to the previous m value is sufficient. See [39, page 371] for method that does not require square root of $N - m^2$. This method is inferior to the continued fraction method and the quadratic sieve method which also rely on Legendre's congruence. Modifications to this method have been suggested by Sherman Lehman [69].

The quadratic sieve method [57, 58] again uses quadratic residues to solve Legendre's congruence and find factors of N. A sieving procedure is used to determine if solutions to $x^2 - N$ can be factored over a preselected set of numbers. The two solutions, a_1 and a_2, of $(x - \lceil\sqrt{N}\rceil)^2 - N \equiv 0 (\mathrm{mod} p)$ are found for each p in

the factor base and the sequences of values $a_1 + kp$ and $a_2 + kp$ are sieved to find a perfect square.

A $\rho + 1$ variant of Pollard's $\rho - 1$ method [65] uses Lucas sequences and performs well when $\rho + 1$ can be factored into small primes where ρ is some factor of N. Pollard's ρ method [65] determines factors, p, by finding the period of an integer sequence, recurrent modulo p (see also Brent's modification [65] and the Monte-Carlo method [39]).

The more recent methods, which are providing the best results for the larger factorisation attempts (approximately 100 digits), are the multiple polynomial quadratic sieve method [70], Lenstra's elliptic curve method [58] and the number field sieve [43]. Their implementation was beyond the scope of this thesis.

7.3 Primality

Determining the primality of factors is an important part of the factorisation process. Each of the factorisation methods commonly used find only one factor and many of the methods do not guarantee that the factors found are prime. To completely factorise an integer into its prime factors usually requires the factorisation algorithms to be applied recursively to the factors found by previous instances. This process is continued until all constituent factors are prime.

Primality can be determined indirectly from the elimination of all possible factors. Some factorisation methods, such as trial division and Pollard's $\rho - 1$ method, have upper bounds, which when reached tell us for certain all factors have been found, but these bounds are usually too high to be used as a practical method for determining primality. Other factorisation methods have no such end point and will either never terminate if asked to factor a prime or they will fail but the failure does not indicate the integer is prime. For this reason factorisation methods should only be applied to numbers which are known to be composite.

Since in primality recognition we are only interested in determining the existence of factors and not the determination of their exact values, their are alternatives to attempting factorisation. Primality tests can be applied which do not prove primality and can fail in very rare circumstances but which can be very efficiently implemented and perform well in most cases. Primality tests work on the premise that if a number is prime it obeys certain conditions. Any numbers which fail the tests are known to be composite but numbers which pass the tests

are not necessarily prime. Provided the number of failures is small the ability to quickly determine primality with near certainty can be useful as a filter to the more computationally intensive primality proving methods.

Rigorous primality proofs for N also exist but they are more involved and typically require the factorisation of $N-1$ or $N+1$. More recently primality proving algorithms have been developed [65, page 138] which have nearly polynomial running times but which are relatively difficult to implement. See [16] and [17] for further details.

7.3.1 Primality Testing

Primality tests are usually based on the following converse of Fermat's Theorem (see Theorem 7.1).

Theorem 7.2 *If a and N are coprime and*

$$a^{N-1} \not\equiv 1 \pmod{N},$$

then N is composite.

By choosing a set of appropriate values for a and applying Theorem 7.2 many composite integers can be identified. This method has a limitation however that cannot be overcome by testing with a sufficient number of a values. There exist composite numbers N which satisfy $a^{N-1} \equiv 1 \pmod{N}$ for all a satisfying $(a, N) = 1$. These numbers are called *Carmichael numbers* and although few in number they invalidate the use of Theorem 7.2 as a general primality test.

A stronger primality test, which may still fail to correctly identify all composite numbers but has no analogues to the Carmichael numbers, is based on the following theorem.

Theorem 7.3 (Strong Pseudoprime Test) *If, for all positive integers a such that a and N are coprime with $N-1 = 2^k m$, m odd and*

$$a^m \equiv 1 \pmod{N} \text{ or } a^{2^i m} \equiv -1 \pmod{N} \text{ for some } i = 0, 1, 2, \ldots, k-1,$$

then N is prime.

A probabilistic algorithm based on Theorem 7.3 would choose a sufficient number of a values, preferably prime. If, for any a value, the condition of The-

orem 7.3 fails then the number is composite. If the test holds for all a values attempted then the integer can be declared to be prime.

Rabin [61] has proved the following result is the best possible,

Theorem 7.4 *If N is composite then the Strong Pseudoprime test fails to hold for at least three quarters of the integers $a \in 1, \ldots, N-1$.*

The confidence we have in the primality of a number tested with this algorithm will depend on the number of a values used with the number of failures at most $(\frac{1}{4})^k$ for k values of a.

For composite numbers the primality test can be terminated as soon as the primality condition is not satisfied for some value of a. Prime numbers will of course satisfy all a values tested and the algorithm will not terminate until all values of a are checked.

In conjunction with the Strong Pseudoprime test an initial check for small factors is also recommended. The likelihood of a number having a small factor is high enough to check by trial division for this possibility.

7.3.2 Primality Proving

The primality testing algorithms discussed above can correctly determine the primality of most numbers but the possibility of failure can limit their usefulness. For applications where primality must be determined beyond a doubt a primality proving method is required.

A converse of Fermat's Theorem originally proposed by Lucas and also D.H. Lehmer [65] gives us a means of determining primality without failure.

Theorem 7.5 *If there exists an integer a such that $a^{N-1} \equiv 1 \pmod{N}$ and $a^{\frac{N-1}{q}} \not\equiv 1 \pmod{N}$ for all $q \mid N-1$ then N is prime.*

By checking a sufficient number of prime a values until the condition of Theorem 7.5 holds the primality of N can be determined. The number of a values that will satisfy Theorem 7.5 when N is prime is approximately $6N/\pi^2$ [22] so only a few choices of a are usually needed to determine primality.

The condition that a single a can be found to satisfy Theorem 7.5 can be relaxed [39, page 395]. Provided a suitable value of a can be found for each prime factor q of $N-1$ then N is prime.

An algorithm, based on Theorem 7.5, requires the factorisation of $N-1$. If the factorisation of $N-1$ is easily obtained, as is the case with numbers of the form 2^m+1 or 7.3^m+1 for example, then proving primality is straightforward. However, for the general case the factorisation of $N-1$ can be difficult to determine making the determination of the primality of N by this method impractical. A major improvement to the general application of Theorem 7.5 is possible as the complete factorisation of $N-1$ is not strictly required [39, page 397]. The factorisation of $N-1$ into a product of primes p_i and a cofactor r such that $r < \sqrt{N-1}+1$ is enough to prove the primality of N if, for every p_i, there is an integer a such that $a^{N-1} \bmod N = (a^{(N-1)/p_i}, N) = 1$.

An alternate method based on the factorisation of $N+1$, described in [39] and attributed to Lucas and Lehmer, could be used or a partial factorisation of $N-1$ and $N+1$ can be combined to further enhance the procedure [9].

If N is composite Theorem 7.5 will fail for all choices of a tried but this failure cannot be used as a proof that N is composite. Applying a primality test such as Theorem 7.3 in conjunction with a primality prove is still necessary to prove a number is composite.

Possible Parallelism: There are several obvious places in which the required operations for primality testing and primality proving can be performed concurrently. By evaluating the primality test and primality proof in parallel the time involved in a successful primality test can be nullified. The set of a values to be tested in Theorems 7.3 and 7.5 can be applied in parallel. When a value for a is found that fails the Strong Pseudoprime Test or satisfies Theorem 7.5 then the compositeness or primality of the number is known and the other concurrently executing tasks can be terminated. Of course, for the primality proof algorithm based on Theorem 7.5 the factorisation of $N-1$ can use a parallel method to further increase the parallel activity.

7.4 A Parallel Factorisation Package

A sequential version of the trial division method of Section 7.2.1 has been included in the implementation because of its usefulness in finding small factors. The probability of small factors in a given number is higher than finding large factors [65], so searching for small primes is worthwhile and can reduce the size of

```
;;; factorise N where any small factors (< 1009) have been removed.
(define (Factorise N)
    ;;; begin primality proving concurrently
    (set! pprove (remote #T PrimeProve N))

    ;;; test for primality in parallel with slower proof
    (if (PrimeTest N)
        ;;; if prime test true, wait for prime proof and return prime if true
        (if (touch pprove)
            (return N))

    ;;; if primality testing fails, terminate unnecessary prime proving task
    (kill pprove)

    ;;; check for perfect power
    (set! roots (check-nthroots N))
    (if roots (Factorise roots))

    ;;; evaluate the factoring methods in parallel, wait for the first method to complete
    ;;; On succesfully finding a factor, terminate remaining methods
    (set! factor (parallel-or (TrialDivision N)
                              (PollardRho    N)
                              (ContinuedFraction N)))
    (set! cofactor (/ N factor))

    ; remove common divisors of factor and cofactor
    (make-coprime factor cofactor)

    (return (append (remote #T Factorise factor)
                    (remote #T Factorise cofactor)))
)
```

Figure 7.1: Pseudo code of top-level factorisation routine

the remaining number to be factored.

A parallel version of trial division has not been implemented as the exhaustive search approach adopted is less effective as the trial divisors increase in size and the number of primes becomes less frequent. The processing resources that would be required can be more effectively applied to other factorisation methods.

Similarly, only a sequential version of Pollard 's $\rho - 1$ method has been implemented due to the low expectation of success for using multiple instances when compared to using other methods.

7.4.1 Repeated Factors

An integer is completely factored when it can be written as a product of powers of primes. Let N be the integer we wish to factor. Then,

$$N = \prod_i p_i^{\alpha_i}, \tag{7.8}$$

where p_i are distinct prime numbers, is the complete factorisation of N.

Most methods for factorisation proceed by finding two factors, m and n such that $N = mn$. By recursively applying the factorisation method to the factors found, further factors can be extracted. This process can be continued until the number to be factored is prime. The complete factorisation of N can then be constructed from the sub-factorisations performed.

Although the above approach will work, no consideration is made of efficiency when N consists of primes to some power other than 1. In these cases there can be duplicity in the factorisation process that can be removed. For example, consider the case where N is the perfect square $N = m^2$ for some composite number m. Assuming the first application of factorisation to N determines m as a factor, then to find the complete factorisation the straightforward approach would attempt to factor m twice, once as the factor and once as the cofactor.

To avoid instances of the above inefficiencies, the factors of N should be checked for common factors and further factorisation should be performed only once for each distinct factor. Common factors occur when factors are not coprime and the greatest common divisor of the two factors is not 1.

Again let N be partially factored into integers m and n. Let $(m, n) = g_1$ represent the greatest common divisor of m and n. If $g_1 \neq 1$ then $\exists m_1, n_1$ such that $m = g_1 m_1$, $n = g_1 n_1$ and $(m_1, n_1) = 1$. We can now write N as $N = m_1 g_1^2 n_1$. Although m_1 and n_1 have no common divisors it is still possible that $(m_1, g_1) \neq 1$ or $(n_1, g_1) \neq 1$. Let $(n_1, g_1) = d_2$ where $d_2 \neq 1$ and let $n_1 = d_2 n_2$ and $g_1 = d_2 g_2$, then $n_1 g_1 = n_2 d_2^2 g_2$ and N can now be written as

$$N = n_2 d_2^3 g_2^2 m_1. \tag{7.9}$$

Because m_1 and n_1 are coprime, m_1 will be coprime to any divisors of n_1 so that $(m_1, n_2) = 1$ and $(m_1, d_2) = 1$. Having found factors of g_1, namely d_2 and g_2, it is still necessary to check for common divisors between g_2 and m_1 as g_1 and m_1 were

not necessarily coprime. It is also necessary to check (n_2, d_2) and (g_2, d_2).

We can continue finding common divisors between distinct pairs, in the manner described, until no more divisors can be found. At this point $n_1 g_1$ can be written as

$$n_1 g_1 = \prod_i q_i^{\beta_i}, \tag{7.10}$$

where the q_i are coprime. Of these q_i all but one can be guaranteed to be coprime to m_1 as all but one of the q_i is a factor of n_1 and $(n_1, m_1) = 1$. It is therefore only necessary to check for common factors between m_1 and this single q_i divisor to construct a set of coprime factors of N.

These factors, when raised to the correct power, will produce the partial factorisation

$$N = \prod_i q_i^{\beta_i}, \tag{7.11}$$

where the q_i are coprime. The factorisation of each q_i will not produce any common primes and no duplication in the factoring will occur.

For example, consider the factorisation of $N = 104810400 = 2^5\ 3^3\ 5^2\ 14557$. Assume a partial factorisation finds a factor 360 so that $N = 360\ 873420$. The greatest common divisor of 360 and 873420 is 60 giving us $N = 6\ 60^2\ 14557$. We then look at $(6, 60) = 6$ giving us $6\ 60 = 1\ 6^2\ 10$ and $(6, 10) = 2$ for $6\ 10 = 3\ 2^2\ 5$. Then as $(3, 2) = 1$ and $(5, 2) = 1$ we can rewrite 6 60 as the pairwise coprime factors $2^3\ 3^2\ 5$. Of the coprime factors for 360 the only possible factor not coprime to 14557 is 5 and as $(5, 14557) = 1$, N can be written as

$$N = 2^5\ 3^2\ 5^2\ 14557, \tag{7.12}$$

where the factors are pairwise coprime. Normally a complete factorisation is not directly attainable. This example succeeded in factorising fully because the prime factors in the first factor found, $360 = 2^3\ 3^2\ 5$, were all present to different powers allowing the greatest common divisor to readily extract the final primes.

Checking for small prime factors is usually attempted before the number is factored by the more general purpose factorisation method.

7.4.2 Perfect Powers (when $N = q^\alpha$)

The factorisation of numbers of the form $N = q^\alpha$, where $q < N$ and $\alpha \in \mathbb{Z}^+$, could be found significantly faster if the factor q could be determined.

In order to use the gcd method on a number of this form a partial factorisation of the integer is first required. Even when N has been factored into two factors m and n where $m = q^{\alpha_1}$, $n = q^{\alpha_2}$ and $\alpha_1 + \alpha_2 = \alpha$ the gcd method will only find factors $q^{(\alpha_1,\alpha_2)}$. For example, if $N = q^4$ and the factor q^2 is found then the gcd method will fail to discover the factor q. The power of q in the partial factorisation must be coprime to the power of q in the cofactor for the gcd method to discover q as a factor.

For specific values of α we can attempt to find an integer q such that $N = q^\alpha$ using numerical root-finding techniques. Finding the positive real root of $x^\alpha - N = 0$ and checking that this root is integral will find the q required. By using small primes such as $2, 3, 5$ and 7 for the value of α the most common occurrences of perfect powers can be found. Any power which can be completely factored over the α used can be reduced to the factor q by repeatedly applying this method. For example, for $N = q^{45}$ and checking roots for $\alpha = 2, 3, 5$ the method will first discover that N can be written as $N = n_1^3$ where $n_1 = q^{15}$. By reapplying the method to n_1 we can also find that $n_1 = n_2^3$ and that $n_2 = q^5$.

7.5 Analysis

7.5.1 Parallel vs Sequential

The performance benefits of this parallel implementation over an equivalent sequential version are difficult to quantify. The parallel implementation relies heavily on the concurrent evaluation of sequential algorithms. The use of sequential algorithms in parallel cannot be easily emulated in a uniprocessor environment. Any attempts to do so would require significant fine tuning of the sequential method in an effort to give an appropriate slice of the processing time to each algorithm. The performance of the sequential method will be very dependent on this scheduling. Judging the relative priorities to assign each factorisation method for an arbitrary integer, without any additional information as to the number's structure, can only be performed heuristically based on general assumptions about the relative performance of each algorithm. The non-uniformity in the number and size of an integer's prime factors, means there are many pathological cases that a heuristic treatment would handle badly. Any direct comparison between this parallel implementation and a multitasking sequential implementation would therefore be misleading.

	Factors	Test	Prove
1	328006342451 . 165768537521	157	158
2	765041185860961084291 . 811	2	8
3	5589928324657 . 78952752017	7	8
4	3750587875813 . 39899809 . 30197	109	110
5	13055351529193 . 514851898711	4	5
6	701788271390597999 . 37243 . 1283	14	19
7	237649652991517758152033 . 1697	4	11
8	1512348937147247 . 78952752017	4	7
9	92820900823386202843255 93 . 11827	7	96
10	3511321676267937799 . 281971 . 113779	11	15
11	574933708017116710865237 . 256057	9	17
12	737539985835313 . 2540413 . 300427 . 683	6	7
13	2778942057555023489 . 218568437 . 14557	889	893
14	263130836933693530167218012159999999 . 11173 . 937	11	314

Table 7.1: Comparison of factorisation times using primality testing and proving

The same reasons that make the parallel implementation difficult to compare to a sequential version highlight the benefits of a parallel implementation that provides for speculative computing in a dynamic manner. Since the method which will provide a factorisation in the shortest time for a given number is difficult to predetermine, the ability to perform each method concurrently with no tradeoff in performance is very desirable. For numbers with more than two factors, different methods may be best suited for finding each of the factors, making the best choice of algorithm dynamic, even for a specific integer.

7.5.2 Proving vs Testing

Table 7.1 compares the factorisation times, for a set of medium sized integers from [22] using primality testing and primality proving. All times are given in seconds. The examples were evaluated on a fully-connected eight-process configuration. The integers to be factored range between 23 and 43 digits. All examples factorised completely using primality testing.

The differences between the times for factorisation using primality testing and proving vary considerable between examples. For examples where the time to factorise is small but the time to prove primality is large, typically when they have a large and small prime factor, using primality testing is significantly faster (cf.

x	Factors of $10^{30} + x$	Test	Prove
1	4188901 . 39526741 . 27961 . 3541 . 9901 . 101 . 61	18	18
2	45802327746425579083 . 84623843 . 43 . 3 . 2	788	918
3	537684419034673655130289 . 1859827	279	307
4	67021328093 . 18737561 . 1267985441 . 157 . 2 . 2	251	263
5	1399606163 . 10085210079364883 . 4723 . 5 . 3	336	364
6	758405810021 . 24032284101871 . 3919 . 7 . 2	695	796
7	10238844796821566353 . 387727 . 251897	104	111
8	45251558227 . 1666666667 . 485899 . 379 . 3 . 3 . 2 . 2 . 2	34	35
9	57613642910641239845595437 . 1021 . 17	5	15
10	154083204930662557781201849 . 59 . 11 . 5 . 2	2	9
11	6275371 . 53117709428388111767947 . 3	1391	1611
12	4635491895871499269567 . 2251 . 97 . 19 . 13 . 2 . 2	4	11
13	1917862968030234283 . 10641095639 . 7 . 7	435	472
14	487083044419 . 342173000223132751 . 3 . 2	710	743
15	139031879314767479609237 . 2659 . 541 . 5	5	40
16	616555753529 . 914813630641 . 3821 . 29 . 2 . 2 . 2 . 2	30	34
17	896295878024886551349239 . 2339 . 53 . 3 . 3	5	14
18	500000000000000000000000009 . 2	3	37
19	156534074818411 . 26737 . 28409 . 3323 . 2531	15	22
20	313378923550840603 . 445847 . 17041 . 7 . 5 . 3 . 2 . 2	112	117

Table 7.2: Comparison of factorisation times using primality testing and proving

examples 2, 9 and 14 in Table 7.1). For other examples, where the factorisation is the most time-consuming step, as when factors are of approximately equal size, there is relatively little difference in processing time (cf. examples 1, 4 and 13 in Table 7.1).

7.5.3 Parallel Continued Fraction Method

The parallel implementation makes strong use of the speculative properties of the Scheme^n design. In addition to this the continued fraction method, the main factorisation method implemented, which should provide the best performance in the majority of cases, has been enhanced to perform concurrently. By using multiple processors in the factoring of integers over a factor base (see Section 7.2.3), the speed at which the algorithm evaluates is improved.

Factoring a single integer over a factor base does not require a great deal of computation. The time for the algorithm is based on the high number of attempts that are required before a suitable factorisation can be found. Attempting to concurrently factor these values by sending individual numbers to other processes is inefficient. The overhead involved in constructing the remote task, which requires, as arguments, the factor base of large integers, makes such low-level concurrency inadvisable.

A more practical method combines together, as a single remote task, multiple numbers for factorisation, greatly reducing the overhead per integer. By determining the appropriate number of integers to assign to each remote task so that the overhead is small in relation to the effective computation but the tasks are not too large as to significantly delay further processing when successful factorisations are found, the best performance is found. For the implementation, 100 numbers per remote task were found to be adequate.

The factorisation step is carried out in tandem with the generation of the numbers to be factored. The generation task is inherently sequential and can only be performed by a single process. By determining the correct ratio of processes to assign to the more computationally involved factorisation, the time to factor the integers can be overlapped with the integer generation. An exact overlap will effectively remove the factorisation time from the evaluation significantly improving the overall performance of the algorithm. Figure 7.2 illustrates how the integers Q_i can be continuously generated by one process while several other processes are used to factorise the generated values at a rate that matches that of the generating

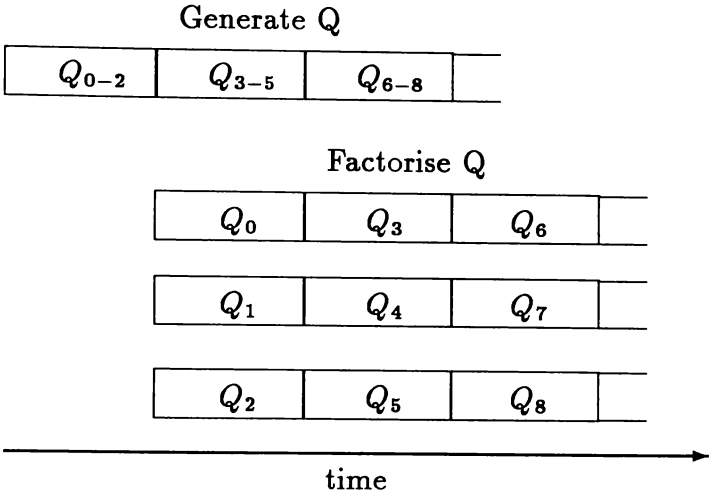


Figure 7.2: Parallel evaluation of Morrison and Brillhart's Continued Fraction Method

process. The limiting factor of the factorisation is the speed at which the sequential sequence of Q_i values can be generated.

The time required to factor a set of generated numbers of a given factor base is dependent in the size of the factor base and the size of the numbers involved. The factorisation process also uses an early-abort strategy which can significantly affect the time required for the attempted factorisation. The variability introduced in the factorisation times with early abortion is less pronounced when the the time is averaged over a group of numbers. If the number of processes required to speed up the factorisation to match the number generation speed is greater than the number of available processes the generating task will become idle while the insufficient number of processes perform the factorisations. Any idle time in the generation task will increase the overall factorisation time. To remove idle CPU cycles the generating task can instead generate and factor the next set of numbers in the continued fraction sequence. The size of the set of numbers should provide just enough additional work to the generating process for the factorising tasks have completed.

For 40 digit numbers, an adequate ratio of factorising tasks to generating tasks was found to be approximately 5:1. For the eight-processor configuration, used for the majority of this analysis, this ratio coincidentally allows all processors to be fully utilised; six processors are involved with the continued fraction method while the remaining two processes concentrate on the trial division and Pollard $\rho - 1$ methods. With this ratio there was still a small number of unused CPU cycles in the generating task, enough to generate and factor approximately 25 numbers. As each of the five factoring tasks were each assigned sets of 100 numbers to factor,

CFRAC Tasks	Time	Speedup
1	47h 8m 55s (rerun)	1.0
3	17h 59m 17s (rerun)	2.6
6	10h 3m 34s	4.7

Table 7.3: Parallel Continued Fraction Performance for $10^{40} + 5$

x	Factors of $10^{40} + x$	Test	Prove
1	197210611666467174983596 81 . 5070721 . 5882353 . 17	17	
2	506774895097 . 651296280594275489 . 5049577699 . 3 . 2	9930	
3	5026110644799734621357954574011992 3 . 661 . 43 . 7	4	
4	–	> 64 hrs	
5	197348950137167 . 20755039827380417341 . 3463 . 47 . 5 . 3	64757	
6	1559965879 . 338004571812645299 . 97759865719 . 97 . 2	11362	
7	39550638490383277927569929 . 4136239 . 375019 . 163	1295	
8	13782909299 . 124853279104459 . 80709870849929 . 3 . 3 . 2 . 2 . 2	64678	
9	331887523968626579430739018230 97 . 13441 . 773 . 29	8	
10	388847808493 . 1058313049 . 6397 . 216451 . 859 . 157 . 13 . 13 . 11 . 7 . 5 . 2	55	

Table 7.4: Example factorisation times for 40 digit numbers

the ability to factor an additional 25 numbers in the same time represents a speed increase of five percent.

As a specific example of the performance of the parallel continued fraction algorithm, Table 7.3 gives times for the factorisation of the number $10^{40} + 5$ for different numbers of concurrent tasks used in the continued fraction algorithm. The number $10^{40} + 5$ has several small factors of 3, 5 and 47, found by searching the table of small primes, and a factor 3463 found quickly with the trial division algorithm. The remaining two factors are 15 digit and 20 digit numbers, which are found with the continued fraction algorithm, and require the most time to find. Ignoring the small fraction of time needed to find the small factors, the ideal speedup for the factorisation would equal the number of parallel tasks used in the continued fraction method. The results fall short of the ideal, which is not unexpected, due to remote task overheads. The actual speedup still represents a significant increase in performance. For larger integers and factor bases, the number of concurrent tasks that can be used beneficially will increase providing even better performance increases.

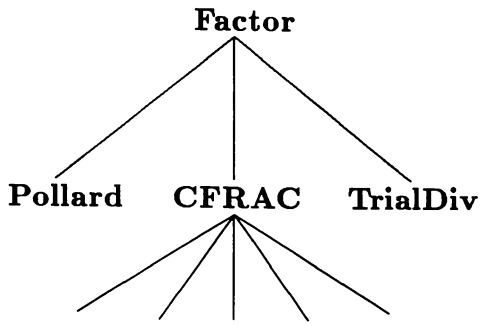


Figure 7.3: Pattern of remote task creation during factorisation

7.5.4 Effect of Process Configuration on Load Balancing

The process configuration can affect the evaluation of a parallel procedure by changing the communication patterns. In particular, the placement of remote tasks can change, modifying the load on each process and the overall utilisation of the system. Changes to the configuration can also cause longer message distances and introduce message contention on highly loaded paths resulting in larger communication delays. The configuration that matches the requirements of the algorithm will provide the best utilisation of available resources. For the factorisation application, Figure 7.3 illustrates the pattern of remote tasks that are created. A configuration that matches this algorithmic structure will perform well. Configurations with too few processes or a communication network that does not provide suitable communications paths will fail to capitalise on the available concurrency.

The asymmetry in remote task activity between the separate factorisation methods can cause some problems in the task assignment. When the three factorisation methods are assigned to processes, as remote tasks, there is no indication as to their future parallel activity. There is an implicit assumption that the parallel requirements of each remote task are equivalent. If the process configuration itself is asymmetrical, the continued fraction method, which requires additional processes, may be poorly assigned to a process with low process connectivity or whose neighbouring processes may be fully occupied. In such situations, manual assignment of remote tasks to processes is required to properly utilise the resources. In a symmetrical configuration these problems can also occur but there is less distinction between alternate locations.

To further illustrate the parallel activity present in the implemented fac-

torisation package, a specific example will be employed. Let

$$\begin{aligned} A &= 156534074818411 \text{ (15 digits)} \\ B &= 15384616923077 \text{ (14 digits)} \\ C &= 727997 \text{ (6 digits)} \end{aligned}$$

be the prime factors of the 34 digit composite number ABC to be factored.

The application of the package to the factorisation of ABC is studied for the 8-process fully-connected, 9-process grid, 8-process ring and 7-process tree configurations. Figures 7.4,7.5,7.6 and 7.7 show how the load of each process varies with time during the factorisation. The number of LWPs active in each process is shown for each configuration, along with the cumulative load of all processes. LWPs which are waiting for other remote tasks to complete but not currently processing are not shown.

The factorisation of the three-factor composite number, ABC, has five distinct phases as the numbers are tested for primality and as each factor is found. Before any parallel activity is invoked by the algorithm, the initial process checks for small integer factors and performs a primality test on the number ABC as described in Section 7.3. This takes a very short time, roughly 3 seconds, for each configuration. The three factorisation methods, trial division, Pollard $\rho - 1$ and Morrison and Brillhart's continued fraction method, described in Section 7.2, are then invoked in parallel. The continued fraction method, for this evaluation, uses four parallel remote task evaluations in addition to the main continued fraction LWP. These sub-evaluations account for most of the activity during the evaluation. The trial division and Pollard $\rho - 1$ methods do not themselves generate any parallel activity.

The first factor A is found first by the Pollard $\rho - 1$ method. For each configuration this takes approximately 69 seconds. The similarity in times for all configurations in finding this first factor is due to the Pollard $\rho - 1$ remote task being assigned to an otherwise unused process in each case. For the ring and tree configurations, there is some doubling of load on some processes which although slowing down the evaluation of these remote tasks, does not affect the evaluation of the successful Pollard $\rho - 1$ method. Once the factor A is found all parallel methods are terminated and the numbers A and BC are factored in parallel. For the prime factor A the factoring process is quickly completed as the prime test succeeds. For the remaining composite number, BC, the three factorisation methods

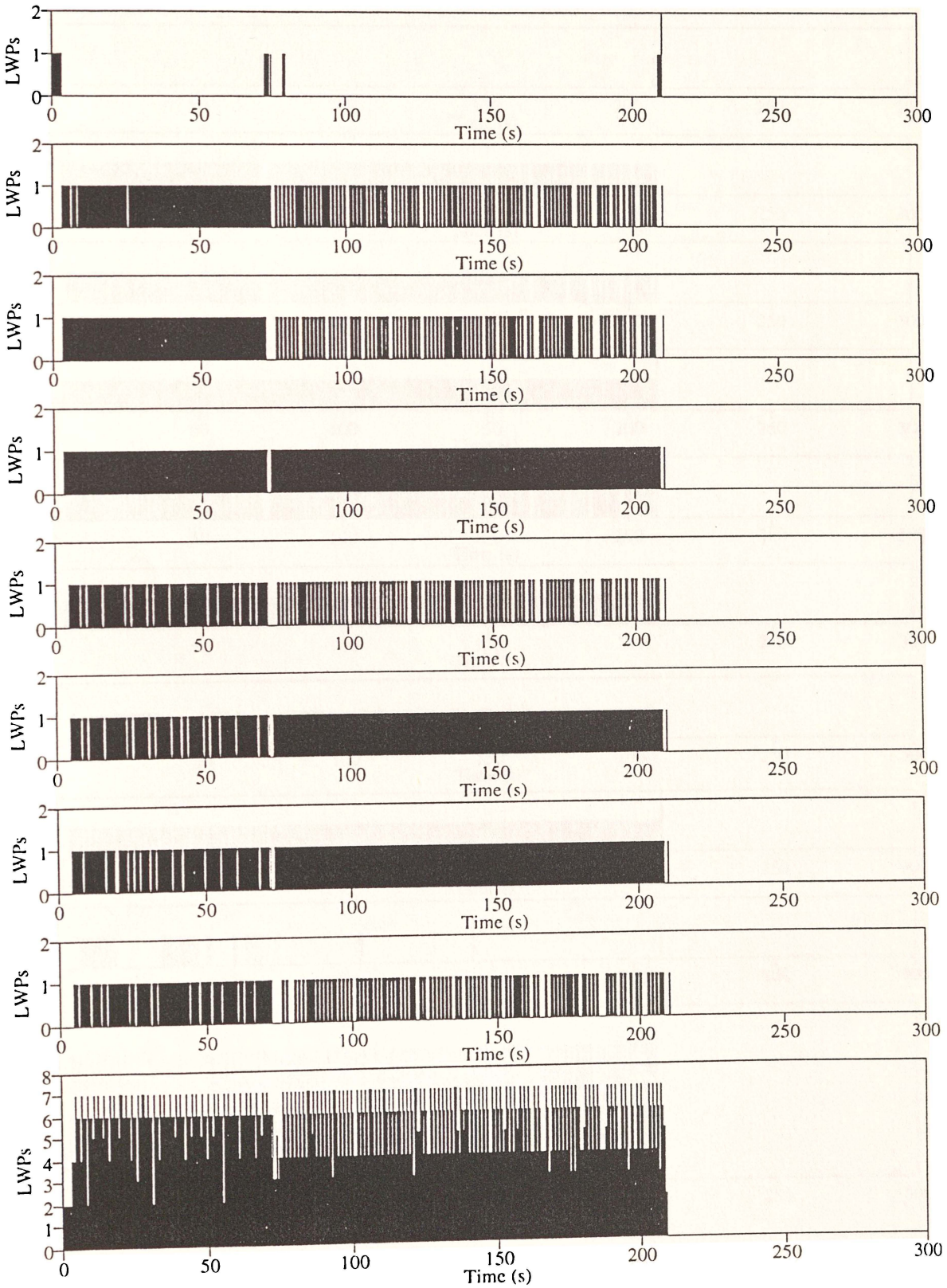


Figure 7.4: Process loads for a fully connected topology

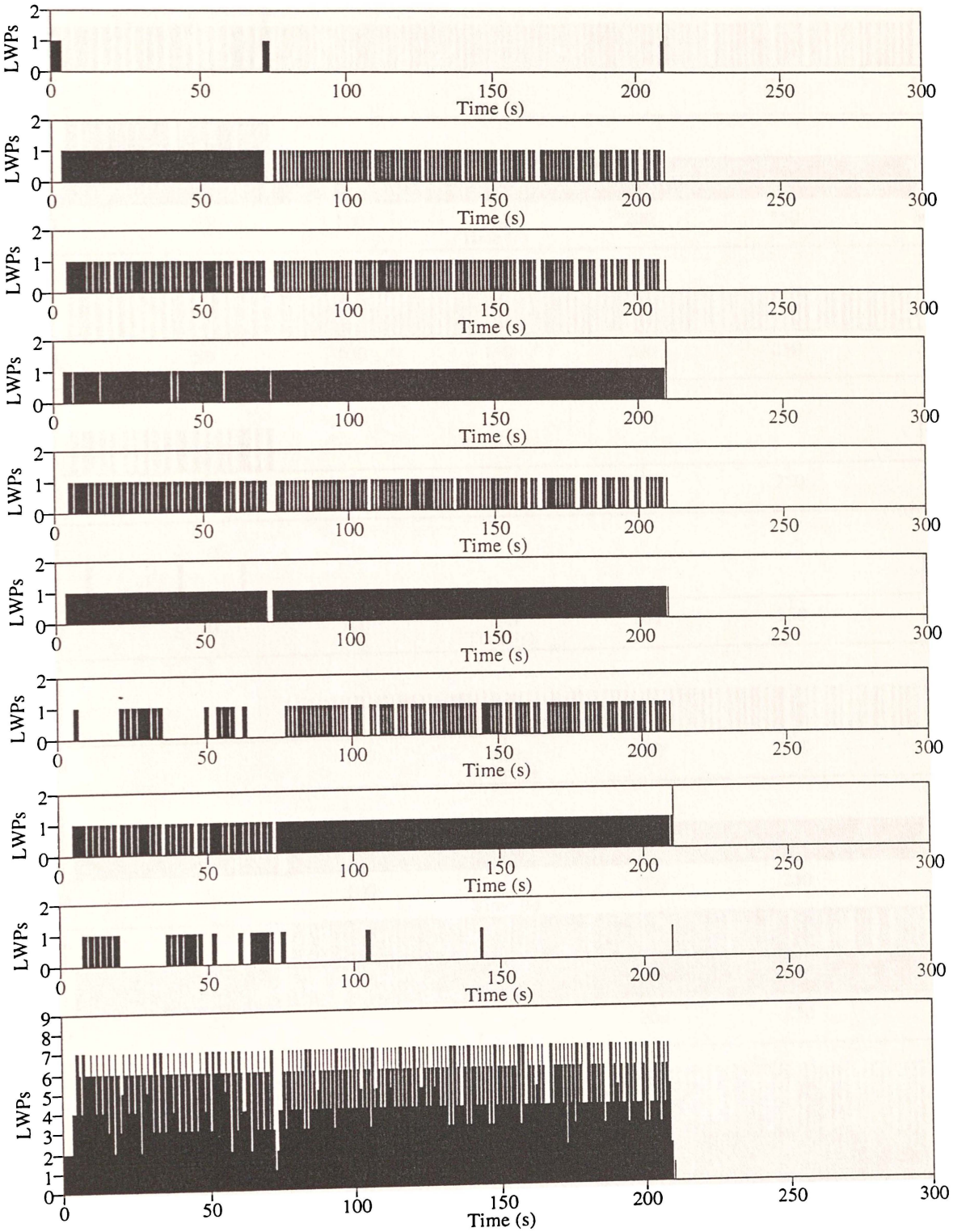


Figure 7.5: Process loads for a grid topology

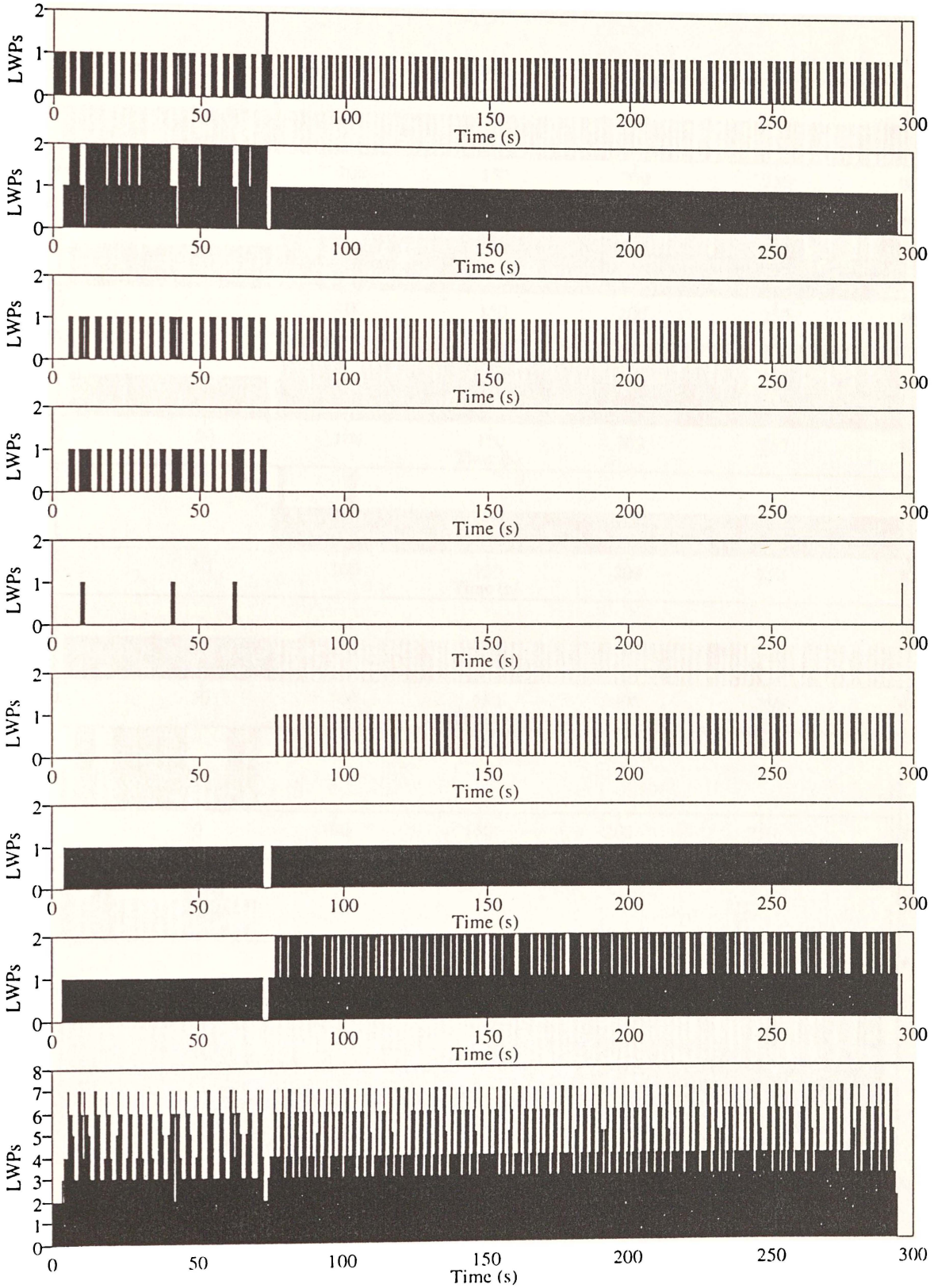


Figure 7.6: Process loads for a ring topology

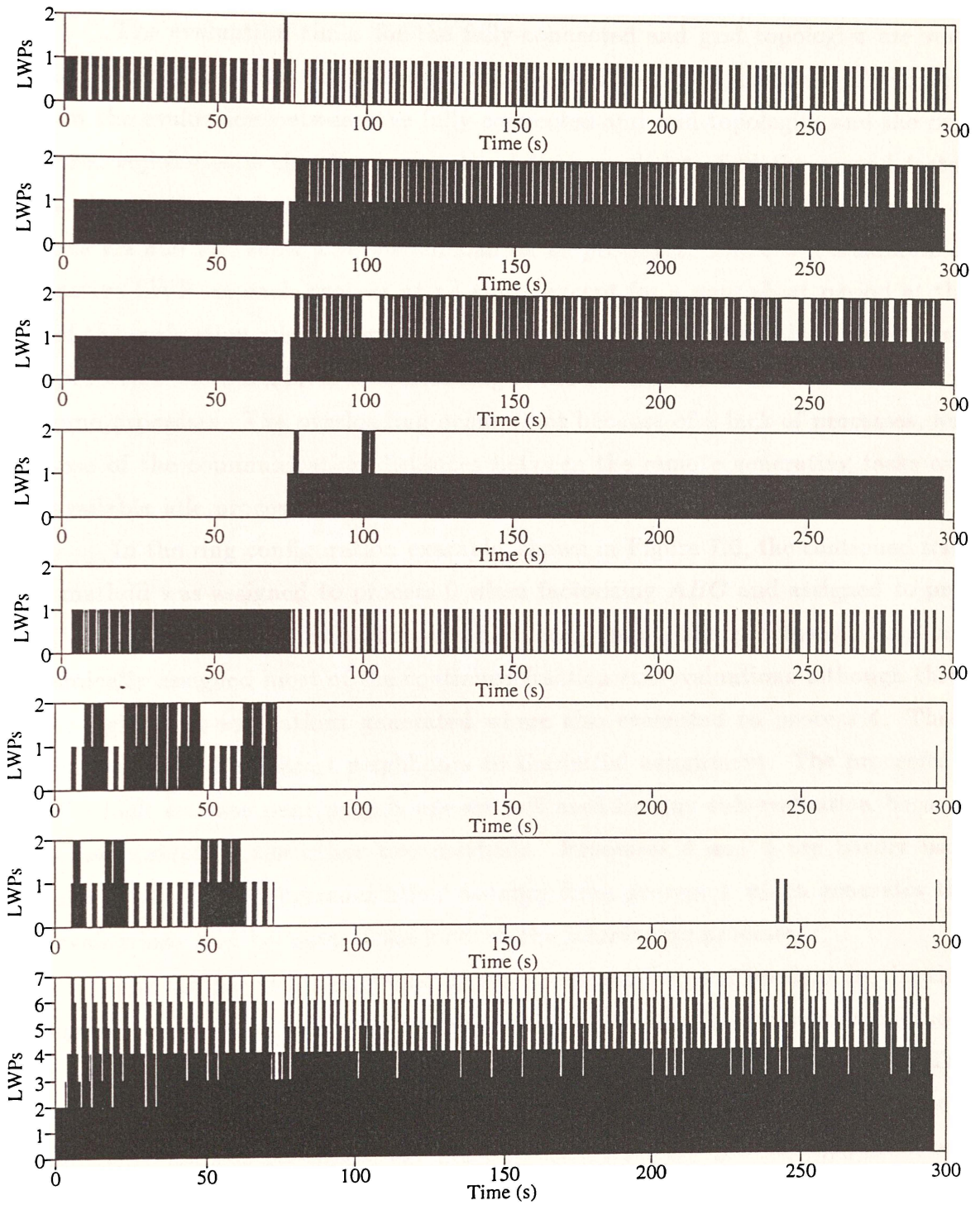


Figure 7.7: Process loads for a tree topology

are again begun in parallel. To find the factor B, the method that succeeds for all configurations is the continued fraction method. The two prime factors are again checked for primality, in parallel, and the factorisation completes.

The evaluation times for the fully-connected and grid topologies are very similar, as are the evaluation times for the ring and tree configurations. The difference in the evaluation between the fully-connected and grid topologies and the ring and tree topologies is due to the length of time needed to find the second factor using the continued fraction method. The fully-connected and grid topologies of Figures 7.4 and 7.6, show a very even load on all processes. There is a maximum of one active LWP on each process at all times except for a very short period at the end of the evaluation where some processes have two active LWPs. For the ring and tree processes there is a level of overloading of the continued fraction sub-evaluations on some processes. The overloading occurs, not because of a lack of processes, but because of the communication distances between the remote generating tasks and the available idle processes.

In the ring configuration example, shown in Figure 7.6, the continued fraction method was assigned to process 0 when factorising ABC and assigned to process 7 when factorising BC. For the first assignment, the processes 0,1,2,3 are dynamically assigned most of the continued fraction sub-evaluations although three of the eighty sub-evaluations generated where also evaluated on process 4. These assignments use the nearest neighbours to the initial assignment. The processes 6 and 7 which are also near neighbours are not used for any sub-evaluation because they are evaluating the other two methods. Processes 4 and 5 are hardly used because of the larger communication distance from process 1 which generates the sub-evaluations and because of the load on the intervening processes.

For the second assignment of the continued fraction method in the ring configuration example, process 7 is assigned the main continued fraction method, with processes 1 and 6 assigned the trial division and Pollard $\rho - 1$ methods. The sub-evaluations, generated by process 7, are assigned to processes 0,2,5 and 7 itself. Again, there are idle processes but their distance from the generating process excludes them from consideration for assignment. In assigning the sub-evaluations, the neighbouring processes 1 and 6 have been bypassed, because of their current loads, and the sub-evaluations given to processes at larger distances from the generating process.

The topology of the tree configuration forces a different load balancing

pattern for the same example (see Figure 7.7). For the first instance of the continued fraction example, process 4, the immediate child node of the root, is assigned the task. The other two methods are assigned to the other branch. The sub-evaluations from the continued fraction method are assigned to processes 0,4,5 and 6 which are all immediate neighbours of process 4. Overloading of these processes occurs, as in the ring example, but rather than overloading process 4, which is evaluating the main continued fraction task, the child node processes of process 4 are given the extra task to evaluate. The child process which evaluates the additional task alternates between the two processes based on their load estimates at each moment. The additional task is never given to process 0 or other processes because its load is perceived to be higher due to the activity on the other branch of processes attached to process 0.

The second use of the continued fraction method in the tree configuration example shows a different way of loading. Because of the continued fraction activity on processes 4,5 and 6 from finding the first factor, each of the three methods is assigned to the other branch of the root process on processes 1,2 and 3. In particular process 1, the immediate child of the root process, is assigned the continued fraction method. Again, overloading occurs as the sub-evaluations are sent to processes 0,1,2 and 4. Process 3 which is currently evaluating the trial division method, is only given three of the three hundred and twenty seven sub-evaluations. Process 6 is also given two sub-evaluations to process.

For the same example, Table 7.5 gives the overall evaluation time, the idle time of each process, the number of remote tasks evaluated be each process and the utilisation of the evaluation for each configuration. All times are given in seconds. Measures such as speed-up and efficiency are difficult to give because of the speculative nature of the factorisation routine. Utilisation is a measure of how much of the available parallelism is used. The parallel factorisation method creates a maximum of seven concurrent tasks for this example. No matter how many processes are available the maximum parallel activity will not exceed seven active LWPs. The utilisation percentage given in Table 7.5 is based on a maximum of seven processes in use at any given time, even though most of the configurations used have more than this number of processes. The utilisation figures ignore the effects of sequential code and assume the algorithm can always maintain activity on seven processes which is not the case. There are many times during the evaluation where the algorithm has less than seven active tasks. The practical values for the utilisation will therefore

	Full		Grid		Ring		Tree	
Processor	Idle	LWP	Idle	LWP	Idle	LWP	Idle	LWP
0	203	(2)	204	(1)	167	(103)	171	(102)
1	74	(84)	75	(84)	8	(20)	7	(85)
2	74	(84)	95	(114)	197	(103)	8	(81)
3	6	(3)	7	(4)	264	(21)	76	(5)
4	86	(120)	94	(114)	290	(4)	164	(84)
5	17	(39)	6	(3)	229	(83)	245	(34)
6	18	(39)	125	(93)	8	(3)	252	(30)
7	87	(119)	26	(34)	7	(85)	-	(-)
8	-	(-)	180	(22)	-	(-)	-	(-)
total idle	565	(490)	812	(469)	1170	(422)	923	(421)
total time (s)	210		210		296		297	
utilisation	75%		73%		58%		56%	

Table 7.5: Evaluation and idle times for different configurations

be significantly less than one hundred percent. The utilisation figures are particular useful in showing the relative characteristics between the given configurations.

The number of LWPs specified in the evaluation of ABC on each configuration is also dependent on the configuration. Each LWP represents a remote task that is evaluated in parallel with its parent. Any remote tasks which were terminated or whose results were required by their parent before they were initiated as LWPs do not generate a separate LWP. As the load on individual processes varies between configurations, the ability to generate and process the remote tasks will also vary. For the example used, the fully-connected configuration has the most even loading and has been able to generate more LWPs than the other configurations. The tree example exhibits significant loading of some processes reducing the rate at which remote tasks are handled and the number of subsequent LWPs.

Knowing the pattern of remote tasks invoked in the parallel factorisation method, the initial assignment of the three factorisation methods is extremely important for good performance. The parallel activity of the continued fraction method and the lack of such activity in the other two methods suggests an assignment policy where the continued fraction method is assigned to a process with many idle neighbours. For the tree configuration used, assigning the continued fraction method, in both instances, to one of the immediate children of the root process and assigning the remaining two methods to leaf processes of the other root branch would provide the best performance. Similarly, the evaluations times for the ring configuration could be improved by ensuring the continued fraction method has as many idle neighbours as possible.

Chapter 8

Summary, Conclusions and Further Work

8.1 Summary

This thesis answers the question posed in Chapter 1 by designing, implementing and analysing Scheme^{n}, a programming language for distributed processing. Chapter 1 introduced the topics of parallel processing and parallel language design and gave a comparison of the Scheme^{n} language to other parallel lisp languages. Chapter 2 presented distributed processing concepts and issues that needed to be addressed in extending a sequential language for concurrency. For each of these issues, the approach chosen for Scheme^{n} was determined.

Chapter 3 defined the syntax and semantics of the extensions introduced to Scheme to provide the programmer with the required concurrency primitives to create parallel tasks, communicate information between the tasks, and synchronise their evaluation. Scheme^{n} does this using a procedure call to dynamically specify procedures for concurrent evaluation, using the standard procedure arguments to pass information to the newly created task, and the procedure result to return values. Variables can also be shared between tasks to allow tasks to communicate during their evaluation. Synchronisation of a parent task with the result of a concurrent task is implicitly resolved through the use of futures, which represent a task's value until it becomes available. Synchronisation of multiple tasks and synchronisation primitives for mutually exclusive access to shared variables are provided. Concurrent tasks can also be explicitly terminated, allowing speculative computation to be used.

Chapter 4 developed the mechanisms needed by Scheme^{n} to provide the

concurrent features of the language. Mechanisms for load balancing and explicit termination are discussed. How the future construct provides synchronisation and communication is explained.

Chapter 5 described the characteristics of the prototype Scheme^n implementation. Together with the hardware and software environment of the implementation, the message passing protocol used to transfer data and maintain the distributed Scheme^n environment is presented. This requires a set of light-weight processes on each processor to distribute and handle the messages.

Chapter 6 used the implemented system to analyse the performance of Scheme^n. The overhead of remote task invocations, communication, and implicit future checking were determined, suggesting a medium to coarse grain of parallelism be adopted. The behaviour of the load balancing mechanism was examined for an number of different connection topologies. It was observed that topology had an effect on task allocation, with topologies that mirrored the algorithm's pattern of remote procedure calls performing better.

Chapter 7 continued the analysis of Scheme^n by implementing a large integer factorisation package and observing its performance. The package applied several levels of parallel activity to the problem of factorisation. Several different factorisation techniques are attempted in parallel, using the speculative ability of the language. Worthwhile parallelism within each method is also exploited. In addition, composite factors are also factored in parallel.

8.2 Conclusions

The programming language, Scheme^n, incorporates many of the features of the thesis question.

No processor has overall control of any aspect of the parallel environment. Memory is distributed amongst processors, such that each processor has exclusive access to its own local memory. When necessary, a shared memory object can be created by any processor, with access synchronised by that processor. Parallel evaluation can also be invoked by any processor, with the allocation of tasks determined by the processors concerned.

Currently, the language has only been implemented on one distributed platform, a network of Transputers. Other environments, such as a network of workstations using a message passing environment such as PVM should also be able to

provide the necessary support. The asynchronous message passing communication model used internally by Scheme^n, requires little support other than point-to-point block message transfer. Other environments, without asynchronous message passing support, should be able to emulate the mimimal communication requirements needed.

The extensions added to the Scheme language syntax are minimal. They clearly support the functional programming model of Scheme and discourage state modification. The future construction is a natural extension of the lazy evaluation strategy, already incorporated into Scheme. No existing syntax has been modified, allowing a Scheme programmer and Scheme source code to easily move to a parallel Scheme^n system. The semantics of Scheme have also been adhered to by avoiding any unnecessary changes to existing semantics. The evaluation mechanism of remote tasks has been hidden from the programmer by using implicit future checking.

The extensions of Scheme^n can be used at a sufficiently low level to provide most general computing needs. Scheme^n has been designed foremost to naturally extend Scheme, and as such, supports the programming style of Scheme above other styles. The inclusion of shared memory objects and the semaphore, as an explicit synchronisation primitive, can be used however to emulate a variety of programming models, but perhaps at the cost of efficiency and clarity.

The extendibility of Scheme^n and its dynamic behaviour when creating and allocating tasks for concurrent evaluation, allow an algorithm to run on many different processor configurations. Changes to processor numbers or interconnections do not require recompilation of the program. Performance can be affected if task numbers and creation patterns do not match the hardware configuration.

Overall, Scheme^n incorporates and extends many features of other parallel lisps in a coherent framework. The future construct has been extended to include lazy and shared variables. These combine to provide support for a wide variety of programming models. In particular, lazy and shared procedures provide a novel approach to distributed data. The automatic features of load balancing and the implicit synchronisation of futures also reduce the burden on the programmer, while the ability to explicitly control these features allows the programmer to maintain tight control when necessary. Load balancing and all other mechanisms have also been designed to minimise synchronisation and avoid centralised control. This specifically supports implementation on larger systems with moderate to high communication overhead; common properties of today's readily available computer installations.

8.3 Further Work

The prototype Scheme^n implementation does not include all features of the design. The use of shared variables has not been needed in developing the large integer factorisation package (although they could be useful) and for that reason they have not been implemented. Similarly, the use of lazy futures and the distribution of futures between processors has not been implemented. A complete implementation of Scheme^n is still forthcoming. Implementations of Scheme^n on other platforms would also help remove any unforeseen dependencies on the Transputer environment and perhaps make Scheme^n more accessible to others. The ability to connect different Scheme^n processes together in a software heterogeneous environment, which was a goal of the thesis that was not realised, would provide more efficient use of resources and allow a more modular set of components to be used together. The introduction of heterogeneity requires some form of registration of the unique attributes of each process.

Many of the implicit future checks performed by the current implementation are not needed and introduce a sometimes excessive overhead. In particular, sequences of operations on non-future objects will repeatedly perform future checks on the same object. With some compiler analysis of the use of futures, it is hoped that most redundant future checks can be removed.

While the dynamic and distributed creation and allocation of remote tasks in Scheme^n is a central advantage of the language, the lack of some method for distinguishing the importance of tasks can lead to inefficient use of resources when excessive numbers of remote tasks are generated. Some priority scheme may need to be investigated to provide an additional level of control over which tasks are evaluated and which are rejected.

Further investigation into the load balancing mechanism is warranted. Inaccuracies in system load estimates could significantly reduce the efficiency of the system. More load messages may be required between processes when changes in neighbouring loads significantly affected a processes load estimate, especially when the last transmitted estimate is higher or lower than the estimates of other processes. Additional messages may also be sent when no messages have been received or sent over a certain period. The lack of activity may be due to inaccurate load estimates hiding available processes from selection. The improvement in estimates, however, must be weighed against the increase message traffic that would be required.

Some method may be required to transmit system loads over a wider area than just neighbouring processes, especially when processes are idle. A wider range of load estimates will improve the rate at which estimates adjust to actual load changes.

An improved averaging formula could be used to take into account information such as the accuracy of the system estimate, the age of estimates and the nature of the system configuration. Improvements of this type woud require more information to be gathered and processed increasing overheads, and perhaps memory requirements, and reducing a processes usable processing capabilty. Information such as the system configuration, while being beneficial in determining the relative importance of system estimates and improving remote task placement, can conflict with design goals such as distributed control and localised knowledge.

Bibliography

[1] H. Abelson and G. Sussman, *Structure and Interpretation of Computer Programs*, The MIT Press, McGraw-Hill, 1985.

[2] G.S. Almasi and A. Gottlieb, *Highly Parallel Computing*, Benjamin-Cummings, 2nd edition, 1994.

[3] F. André, D. Herman, and J.-P. Verjus, *Synchronization of Parallel Programs*, MIT Press series in scientific computation, 1985.

[4] K.E. Atkinson, *An Introduction to Numerical Analysis*, Wiley, 1978.

[5] H.G. Baker, Jr., List Processing in Real Time on a Serial Computer, *Comm. ACM*, **21**(4), April 1978, 280–294.

[6] J.F. Bartlett, SCHEME->C a Portable Scheme-to-C Compiler, *Technical Report WRL Research Report 89/1*, Digital Equipment Corporation Western Research Laboratory, January, 1989.

[7] G. Bell, Ultracomputers: A Teraflop Before Its Time, *Comm. ACM*, **35**(8), August 1992, 26–47.

[8] N. Berrington, P. Broadbery, D. De Roure and J. Padget, EuLisp Threads: A Concurrency Toolbox, *Lisp and Symbolic Computation: An International Journal*, (6) **1/2**, 1993.

[9] J. Brillhart, D.H. Lehmer and J.L. Selfridge, New primality criteria and factorizations of $2^m + 1$, *Math. Comp.*, **29**, 1975, 620–647.

[10] P. Brinch Hansen, The Programming Language Concurrent Pascal, *Concurrent Programming*, N. Gehani and A.D. McGettrick (editors), Addison-Wesley, 1988, 73–92.

[11] K.A. Broughan, SENAC: Lisp as a platform for constructing a problem solving environment, *Programming Environments for High-Level Scientific Problem Solving*, P.W. Gaffney and E.N. Houstis (Eds), North-Holland/IFIP, 1992, 351–359.

[12] K. Broughan, W. Schou and D.K. Willcock, Preparing SENAC for Parallel Computing, *Asia-Pacific High Performance Computing Steering Committee Meeting*, NCHC, Hsin Chu, Taiwan, 28 Feb.–1 Mar. 1994.

[13] A. Burns, *Programming in occam 2*, The Instruction set series, 1988.

[14] S. Chowdhury, The Greedy Load Sharing Algorithm, *J. Parallel and Distributed Computing*, **9**, 1990, 93–99.

[15] D.W. Clark, An Empirical Study of List Structure in Lisp, *Comm. ACM*, **20**(2), February 1977, 78–87.

[16] H. Cohen and A.K. Lenstra, Primality testing and Jacobi sums, *Math. Comp.*, **42**, 1984, 297–330.

[17] H. Cohen and A.K. Lenstra, Implementation of a New Primality Test, *Math. Comp.*, **48**, 1987, 103–121.

[18] D.E. Comer and D.L. Stevens, *Internetworking with TCP/IP Volume 3: Client Server Programming and Applications*, Prentice-Hall, 1993.

[19] G. Cybenko, Dynamic Load Balancing for Distributed Memory Multiprocessors, *J. Parallel and Distributed Computing*, **7**, 1989, 279–301.

[20] H.M. Deitel, *Operating Systems 2nd ed.*, Addison-Wesley, 1990.

[21] D.L. Eager, E.D. Lazowska, and J. Zahorjan, Adaptive Load Sharing in Homogeneous Distributed Systems, *IEEE Trans. Software Eng.*, **12**(5), May 1986, 662–675.

[22] S.D. Galbraith, *Factorization and Primality Testing*, University of Waikato, School of Computing and Mathematical Sciences report, 1989.

[23] N. Gehani and A.D. McGettrick (editors), *Concurrent Programming*, Addison-Wesley, 1988.

[24] D. Gelernter, Generative communication in Linda, *ACM trans. on Programming Languages and Systems*, 7(1):80–112, Jan. 1985.

[25] I.D. Graham and T. King, *The Transputer Book*, Prentice Hall, London, 1990.

[26] A. Hać, Load Balancing in Distributed Systems: A Summary, *Performance Evaluation Review*, **16**(2), 17–19.

[27] R.H. Halstead Jr., MultiLisp: A language for concurrent symbolic computation, *ACM Transactions on Programming Languages and Systems*, (4) **7** (1985), 501–538.

[28] C. Hammer and T. Henties, Parallel Lisp for a Distributed Memory Machine, *EUROPAL'90: High Performance and Parallel Computing in Lisp*, Twickenham, London, UK, 12–13 Nov 1990.

[29] R.V. Hanxleden, Load Balancing on Message Passing Architectures, *J. Parallel and Distributed Computing*, **13**, 1991, 312–324.

[30] High Performance Fortran Language Specification, *Fortran Forum*, Volume 12, Number 4, December 1993.

[31] C.A.R Hoare, Monitors: An Operation System Structuring Concept, *Comm. ACM*, **17**(10), October 1974, 549–557.

[32] C.A.R. Hoare, *Communicating Sequential Processes*, Prentice-Hall International Series in Computer Science, 1985.

[33] K. Hwang and F. Briggs, *Computer Architecture and Parallel Processing*, McGraw-Hill, Singapore, 1985.

[34] IEEE Std 1178-1990, *IEEE Standard for the Scheme Programming Language*, Institute of Electrical and Electronic Engineers, Inc.

[35] INMOS Ltd., *The Transputer Applications Notebook: Systems and Performance*, INMOS Ltd., 1989.

[36] G. Jones, *Programming in occam*, Prentice-Hall International Series in computer science, 1987.

[37] R.R. Kessler and M.R. Swanson, Concurrent Scheme, *Parallel Lisp: Languages and Systems Proceedings (Lecture Notes in Computer Science)*, Springer-Verlag, 1989, 200–234.

[38] D.E. Knuth, *The Art of Computer Programming, Vol. 1: Fundamental Algorithms 2nd ed.*, Addison-Wesley, Reading, Mass., 1973.

[39] D.E. Knuth, *The Art of Computer Programming, Vol. 2: Seminumerical Algorithms 2nd ed.*, Addison-Wesley, Reading, Mass., 1981.

[40] D.M. Koorey Willcock, *A Parallel Lisp and Multivariate Polynomial GCD Implementation*, D. Phil thesis, University of Waikato, New Zealand, 1991.

[41] D.A. Kranz, R.H. Halstead, and E. Mohr, Mul-T: A High Performance Parallel Lisp, *SIGPLAN '89 Conference on Prog. Languages Design and Implementation*, Portland Oregon, June 21-23, 1989.

[42] E.V. Krishnamurthy, *Parallel Processing*, International Computer Science Series, Addison-Wesley, 1989.

[43] A.K. Lenstra, H.W. Lenstra, Jr., et.al, The Number Field Sieve, *Proc. 22nd Annual ACM Symposium on the Theory of Computing*, Baltimore, Maryland, May 14–16, 1990, 564–572.

[44] F.C.H. Lin and R.M. Keller, The Gradient Model Load Balancing Method, *IEEE Trans. Software Eng.*, **13**(1), January 1987, 32–38.

[45] H. Lu and M.J. Carey, Load-balanced Task Allocation in Locally Distributed Computer Systems, *Proc. 1986 International Conference on Parallel Processing*, August 19-22, K. Hwang et al. (Eds), 1037–1039.

[46] B.J. MacLennan, *Principles of Programming Languages: Design, Evaluation, and Implementation*, CBS College Publishing, 2nd edition, 1987.

[47] O.A. McBryan, An overview of message passing environments, *Parallel Computing*, **20**, 1994, 417–444.

[48] D.I. Moldovan, A Comparison between Parallel Processing of Numeric and Symbolic Algorithms, *Parallel Algorithms & Architectures*, M. Cosnard et al. (editors) Elsevier Science Publishers B. V. (north-Holland), 1986, 325–333.

[49] M.A. Morrison and J. Brillhart, A Method of Factoring and the Factorisation of F_7, *Math. Comp.*, **29**(129), January 1975, 183–205.

[50] R.B. Osborne, Speculative Computation in Multilisp, *Parallel Lisp: Languages and Systems Proceedings (Lecture Notes in Computer Science)*, Springer-Verlag, 1989, 103–137.

[51] C. Papadimitriou, *The Theory of Database Concurrency Control*, Principles of Computer Science Series, Computer Science Press Ltd, 1986.

[52] Parallel Computing: Special Issue on Message Passing Interfaces, **20**(4), 1994, April 1994.

[53] J.D. Pehoushek and J.S. Weening, Low-cost process creation and dynamic partitioning in Qlisp, *Parallel Lisp: Languages and Systems Proceedings (Lecture Notes in Computer Science)*, Springer-Verlag, 1989, 182–199.

[54] Perihelion Software Ltd, *The Helios Parallel Operating System*, Prentice Hall, 1991.

[55] J.M. Piquer, Sharing Data Structures in a Distributed Lisp, *EUROPAL Proceedings: High Performance and Parallel Computing in Lisp*, November 1990.

[56] J.M. Pollard, Theorems on Factorization and Primality Testing, *Proc. Cambr. Philos. Soc.*, **76**, 1974, 521–528.

[57] C. Pomerance, Analysis and Comparison of Some Integer Factorisation Algorithms, *Computational Methods in Number Theory, Part I*, H.W. Lenstra, Jr., and R. Tijdeman (eds.), 1982, 89–139.

[58] C. Pomerance, Factoring, *Proc. Symposia in App. Maths*, **42**, 1990, 27–47.

[59] M.J. Quinn, *Designing Efficient Algorithms for Parallel Computers*, McGraw-Hill, 1987.

[60] P. Quinton and J.-P. Verjus, Distributed Synchronization of Parallel Programs: Why and How?, *Parallel Algorithms & Architectures*, M. Cosnard et al. (editors) Elsevier Science Publishers B. V. (north-Holland), 1986, 109-125..

[61] M.O. Rabin, Probabilistic algorithms for testing primality, *J. Number Theory*, **12**, 1980, 128–138.

[62] D.A. Reed and R.M. Fujimoto, *Multicomputer Networks: Message-Based Parallel Processing*, MIT Press, 1987.

[63] J. Rees, W. Clinger (Editors), Revised^3 Report on the Algorithmic Language Scheme, *SIGPLAN Notices*, (12) **21** (1986), 37–39.

[64] J. Rees, W. Clinger (eds), Revised^4 Report on the Algorithmic Language Scheme, ACM Lisp Pointers 4, July-September 1991.

[65] H. Riesel, *Prime Numbers and Computer Methods for Factorization*, Birkhäuser, Progress in Mathematics, 1985

[66] O. Roubine and J.-C. Heliard, Parallel Processing in Ada, *Concurrent Programming*, N. Gehani and A.D. McGettrick (editors), Addison-Wesley, 1988, 142–159.

[67] J.A. Sharp, *Data Flow Computing*, Ellis Horwood Limited, 1985.

[68] S. Shen, Cooperative Distributed Dynamic Load Balancing, *Acta Informatica*, **25**(6), August 1988, 663-676.

[69] R. Sherman Lehman, Factoring Large Integers, *Math. Comp.*, **28**(126), April 1974, 637–646.

[70] R.D. Silverman, The Multiple Polynomial Quadratic Sieve, *Math. Comp.*, **48**(177), January 1987, 329–339.

[71] W.R. Stevens, *Unix Network Programming*, Prentice Hall Software Series, 1990.

[72] P. David Stotts, A Comparative Survey of Concurrent Programming Languages, *Concurrent Programming*, N. Gehani and A. D. McGettrick (editors), Addison-Wesley, 1988, 419–436.

[73] V.S. Sunderam, G.A. Geist, J. Dongarra, R. Manchek, The PVM concurrent computing system: Evolution, experiances, and trends, *Parallel Computing* **20**, 1994, 531-545.

[74] Thinking Machines Corporation, *Getting Started in *Lisp*, Thinking Machines Corporation, 1991.

[75] D.W. Walker, The design of a standard message passing interface for distributed memory concurrent computers, *Parallel Computing*, **20**, 1994, 657–673.

[76] H.C. Williams and M.C. Wunderlich, On the Parallel generation of the Residues for the Continued Fraction Factoring Algorithm, *Math. Comp.*, **48**(177), January 1987, 405–423.