CIRCUIT SIMULATION USING DISTRIBUTED WAVEFORM RELAXATION TECHNIQUES

A Thesis
Submitted to the College of Graduate Studies and Research
in Partial Fulfillment of the Requirements
for the Degree of
Doctor of Philosophy
in the
Department of Electrical Engineering
University of Saskatchewan

by

Anant D. Jalnapurkar
Saskatoon, Saskatchewan, Canada
Spring 1998

© Copyright Anant D. Jalnapurkar, 1998. All rights reserved.

National Library of Canada

Acquisitions and Bibliographic Services

395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque nationale du Canada

Acquisitions et services bibliographiques

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence

Our file Notre référence

The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.

The author retains ownership of the copyright in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

L'auteur conserve la propriété du droit d'auteur qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

0-612-27414-4

Canadä

PERMISSION TO USE

The author has agreed that the Library, University of Saskatchewan, may make this thesis freely available for inspection. Moreover, the author has agreed that permission for extensive copying of this thesis for scholarly purposes may be granted by the Professor who supervised the thesis work recorded herein or, in his absence, by the Head of the Department or the Dean of the College in which the thesis work was done. It is understood that due recognition will be given to the author of this thesis and to the University of Saskatchewan in any use of the material in this thesis. Copying or publication or any other use of this thesis for financial gain without approval by the University of Saskatchewan and the author's written permission is prohibited.

Requests for permission to copy or to make any other use of the material in this thesis in whole or in part should be addressed to:

Head of the Department of Electrical Engineering
University of Saskatchewan
Saskatoon, Canada S7N 0W0

ACKNOWLEDGEMENTS

The author would like to express his gratitude and appreciation to his supervisors Prof. H. C. Wood and Prof. Carl D. McCrosky for their guidance during the course of this work. Their help in the preparation of the thesis is thankfully acknowledged.

The author would like to thank members of the research committee, Prof. A. E. Krause, Prof P. Pramanick, Prof. Ram Manohar, Prof. Ron Bolton and Prof. T. S. Sidhu for their valuable comments and suggestions

Help and guidance provided by Dr. Res Saleh and his student Mr. Y. Wen during author's visit to the University Illinois at Urbana-Champagne is thankfully acknowledged.

The author would like thank Dr. Rama Munikoti, Director, Software Engineering and Analysis Laboratory, Nortel, for his encouragement and support. The author would like to thank his friends, Mr. Pramod Dhakal, and Dr. Atul Jain for their help during the course of this research and in the preparation of the thesis.

The author is grateful to his uncle, Mr. Madhav Jalnapurkar, aunt Mrs. Sheela Jalnapurkar for their constant encouragement and support.

Financial support provided by the University of Saskatchewan in the form of graduate scholarship and, Natural Sciences and Engineering Research Council of Canada, in the form of research assistantship is thankfully acknowledged.

Finally, the author would like to thank his wife Mrs. Archana Jalnapurkar for her patience and support.

ABSTRACT

Simulation plays an important role in the design of integrated circuits. Due to high costs and large delays involved in their fabrication, simulation is commonly used to verify functionality and to predict performance before fabrication. Depending upon the level of abstraction used, simulators may be classified as behavioral simulators, register transfer level simulators, gate level logic simulators, switch level simulators and electrical circuit simulators. Gate and switch level simulators provide approximate timing information, however, none of these simulators provide detailed timing information. Electrical circuit simulation is the only tool that provides accurate timing information and performance details. This, however, requires more than three orders of magnitude more computing time compared to gate or switch level simulators The importance and the high computing cost of circuit simulation provides motivation for the development of fast and accurate electrical circuit simulators. These can be achieved by using improved algorithms and high performance computer architectures to run the simulation engines.

This thesis describes analysis, implementation and performance evaluation of a distributed memory parallel waveform relaxation technique for the simulation of MOS VLSI circuits. The waveform relaxation technique exhibits inherent parallelism due to the partitioning of a circuit into a number of sub-circuits. These sub-circuits can be concurrently simulated on parallel processors. In addition, the full window waveform relaxation technique permits exchange of large and infrequent messages among sub-circuits. This feature is useful for parallel implementation on low cost distributed memory machines.

Different forms of parallelism in the direct method and the waveform relaxation technique are studied. An analysis of single queue and distributed queue approaches to implement parallel waveform relaxation on distributed memory machines is performed and their performance implications are studied. The distributed queue approach selected for exploiting the coarse grain parallelism across sub-circuits is described. A distributed queue implementation involves static partitioning and placement of sub-circuits on processors. An algorithm based on the critical path method and an algorithm based on bin packing heuristics are used to partition Gauss-Seidel and Gauss-Jacobi task graphs respectively. Parallel waveform relaxation programs based on Gauss-Seidel and Gauss-Jacobi techniques are implemented using a network of eight Transputers. Static and dynamic load balancing strategies are studied. A dynamic load balancing algorithm is

developed and implemented. Results of parallel implementation are analyzed to identify sources of bottlenecks.

This thesis has demonstrated the applicability of a low cost distributed memory multi-computer system for simulation of MOS VLSI circuits. Speed-up measurements prove that a five times improvement in the speed of calculations can be achieved using a full window parallel Gauss-Jacobi waveform relaxation algorithm. Analysis of overheads shows that load imbalance is the major source of overhead and that the fraction of the computation which must be performed sequentially is very low. Communication overhead depends on the nature of the parallel architecture and the design of communication mechanisms. The run-time environment (parallel processing framework) developed in this research exploits features of the Transputer architecture to reduce the effect of the communication overhead by effectively overlapping computation with communications, and running communications processes at a higher priority.

This research will contribute to the development of low cost, high performance workstations for computer-aided design and analysis of VLSI circuits.

Table of Contents

PERMISSION TO USE i
ACKNOWLEDGEMENTS ii
ABSTRACT iii
TABLE OF CONTENTS v
LIST OF FIGURES viii
LIST OF TABLES ix
1. INTRODUCTION 1
1.1. Circuit Simulation 4
1.2. Parallel Processing 6
1.3. Motivation and Research Objectives 8
1.4. Thesis Outline 10
2. CIRCUIT SIMULATION TECHNIQUES 12
2.1. Formulation of the Equations 13
2.1.1. Branch Relations 16
2.2. Direct Method 19
2.2.1. Time Step Control 23
2.3. Relaxation Methods 26
2.3.1. Linear Relaxation 27
2.3.2. Nonliniar Relaxation Methods 32
2.4. Iterated Timing Analysis 35
2.5. Waveform Relaxation 37
2.5.1. Windowing Mechanism 42
2.5.2. Circuit Partitioning 44
2.6 Summary 47
3. PARALLEL RELAXATION METHODS 49
3.1. Parallel Processing Techniques 49
3.1.1. Shared Memory Systems 50
3.1.2. Distributed Memory Systems 52
3.1.3. KSR Architecture 54
3.2. Efficiency and Speed-up in Parallel Systems 56
3.3. Parallel Direct Methods 58
3.4. Common Parallelism in Relaxation Methods 61
3.5. Parallel Waveform Relaxation 67
3.6. Parallel Iterated Timing Analysis 70

3.7. Static Partitioning Techniques 71
3.8. Placement Techniques 79
3.9. Summary 82
4. PARALLEL WAVEFORM RELAXATION IMPLEMENTATION 84
4.1. Implementation Strategies 85
4.1.1. Parallel Model Evaluation 86
4.1.2. Single Queue Approach 86
4.1.3. Distributed Queue Approach 87
4.1.4. Multicomputer Interconnection Network 88
4.2. Program Structure Chart 89
4.3. Partitioning Algorithms 92
4.3.1. Partitioning Algorithm for Gauss Seidel Task Graph 92
4.3.2. Partitioning Algorithm for Gauss Jacobi Task Graph 94
4.4. Placement Algorithm 95
4.5. Parallel Transient Analysis 95
4.5.1. Process Framework 96
4.5.2. Instrumentation for Performance Measurements 98
4.6. Summary 99
5. DYNAMIC LOAD BALANCING 101
5.1. Limitations of Static Partitioning 102
5.2. Dynamic Load Balancing Methods 103
5.3. Dynamic Load Balancing Algorithm 105
5.4. Implementation of Dynamic Load Balancing 106
5.5. Summary 111
6. RESULTS 112
6.1. Gauss Seidel Method 113
6.2. Gauss Jacobi Method 117
6.3. Analysis of Performance Limiting Factors 118
6.3.1. Communication Overhead 118
6.3.2. Sequential Computation 122
6.3.3. Load Imbalance Overhead 122
6.4. Dynamic Load Balancing 124
6.5. Summary 125
7. SUMMARY, CONCLUSION AND FUTURE WORK 126
7.1. Summary 126
7.2. Conclusions 130

7.3. Future Work 131
REFERENCES 133
A. DEVICE MODEL EVALUATION 141
A.1. Linear Devices 141
A.2. Nonlinear Devices 143
B. NUMERICAL ANALYSIS 146
C. TRANSPUTER AND OCCAM 149

List of Figures

Figure 2.1	A MOS nand gate.	16
Figure 2.2	A MOS transistor in free space.	18
Figure 2.3	A MOS transistor model.	19
Figure 2.4	Flowchart of simulation process	22
Figure 2.5	Waveform properties [a] Latency [b] Multi-rate behavior.	28
Figure 2.6	Relaxation-based techniques.	29
Figure 2.7	A linear circuit considered for partitioning.	46
Figure 3.1	Shared memory interconnection networks.	51
Figure 3.2	Multi-computer interconnection network topologies.	53
Figure 3.3	KSR-1 architecture with a slotted ring for communication.	56
Figure 3.4	Examples of Gauss-Seidel and Gauss-Jacobi partial ordering for one iteration.	64
Figure 3.5	Unrolled (a) Gauss-Seidel (b) Gauss-Jacobi iterations.	66
Figure 3.6	A n -processor network.	75
Figure 3.7	A two-level search tree with two processors.	77
Figure 4.1	Network topology.	89
Figure 4.2	Program structure chart.	91
Figure 4.3	Circuit partitioning, task graph partitioning and placement.	91
Figure 4.4	Data structures.	97
Figure 4.5	Process framework.	99
Figure 5.1	A dynamic load balancing algorithm.	107
Figure 5.2	The sim.remote() (Simulate Remote) procedure.	109
Figure 5.3	The sim.rdy.ckts() (Simulate Ready Circuits) procedure.	110
Figure 6.1	Node voltage waveforms for OPAMP.	114
Figure 6.2	Node voltage waveforms for DECPLA	115
Figure 6.3	Node voltage waveforms for CRAMB.	116

List of Tables

Table 6.1 Benchmark Circuits. 113

Table 6.2 Speed-up for GS Algorithm. 117

Table 6.3 Speed-up for GJ Algorithm. 118

Table 6.4 Variation of Communication Cost. 121

Table 6.5 Aggregrate Processors Idle Time. 123

1. INTRODUCTION

Advancement in integrated circuit technology has been very rapid since its inception in the early 1960s. Integrated circuit technology has pervasively influenced developments in many areas of science and technology. Several application domains such as data processing, consumer electronics, telecommunications, industrial electronics and automotive electronics have witnessed a phenomenal growth, especially in the last decade. This can largely be attributed to the developments in integrated circuit technology.

The economic impact of this technology in the world market has been remarkable. According to 1992 figures, electronics goods had a total market of over $1000 billion, which constituted approximately 10% of the worldwide gross product. Projections for year 2000 indicate that electronics will have a market of over $3000 billion [1][2]. A large sector of this market is driven by advancements in integrated circuit technology and its ever increasing applications.

The complexity of integrated circuits has grown tremendously. In the late 1960s, it was projected that the transistor density of chips would quadruple every three or four years. The actual growth in the complexity of the integrated circuits has exceeded these expectations.

An integration density of up to one million transistors per chip has become the reality and industry is now focusing on the integration of a billion transistors in a single chip. The major contributing factors in the growth of the integration density have been high-resolution lithography, improved reliability in processing silicon wafers, better understanding of system level design issues, and the availability of better computer aided design tools for circuit layout, simulation, verification and testing.

Simulation plays an important role in the design of integrated circuits. Due to high costs and large delays involved in their fabrication, simulation is commonly used to verify functionality and to predict performance before fabrication. Fast and accurate simulation programs have been used to reduce the time to market the products. They provide a competitive advantage in today's rapidly changing marketplace.

Simulation programs have replaced traditional breadboard based prototyping techniques which are commonly used to validate circuits consisting of discrete components. Development of a breadboard based prototype of a modestly sized integrated circuit can be expensive and time consuming. In addition, it may provide grossly inaccurate results due to differences in the behavior of devices and values of parasitic components. Simulation programs also permit evaluation of architectural and design alternatives, during different phases of a development cycle, at a substantially lower cost than build and test methods. Early feedback on the validity of architecture and design can provide large economic gains.

A simulation program represents a circuit to be simulated in the form of an abstract model. The model accepts the primitive elements, and the rules of interconnection and operation as inputs. The output of the simulator describes the predicted behavior and performance characteristics of the circuit.

Different levels of abstraction of the circuit model are used at different stages of the design process. Depending upon the level of abstraction used, simulators may be classified as behavioral (also called algorithmic or functional) simulators, register transfer level simulators, gate level logic simulators, switch level simulators and electrical circuit simulators [2].

Behavioral simulators are used during the initial phase of design to verify important design concepts and algorithms. Behavioral simulators describe digital systems using functional blocks. Examples of applications appropriate for behavioral simulation include validation of the operation of a direct memory access (DMA) controller, or checking a new network protocol for a local area network (LAN).

Register transfer level (RTL) simulators describe integrated circuits using combinational components such as multiplexers, arithmetic units, and sequential components such as counters and registers. RTL simulators have been commonly used for data path design.

Gate level logic simulators use macro-models of logic gates that simulate digital circuits. Gate level simulators provide approximate timing information including the detection of hazards, glitches, and race conditions.

Switch-level simulators are used for logic simulation of MOS digital circuits. These simulators simulate the circuit at the transistor level. The transistors are modeled as gate controlled switches.

Electrical or circuit level simulators describe integrated circuits using devices such as transistors, resistors, capacitors and diodes. A system of equations is formulated using circuit topology, device models, input signals and initial conditions and solved using numerical methods. Circuit simulators provide detailed information about the circuit behavior and performance.

The nature of information provided by a simulator and its execution time depend on the level of abstraction. In general, as the level of abstraction goes down, the amount of computation increases. Behavioral and RTL simulators verify important design ideas and compare architectural alternatives without providing timing information. Gate and switch level simulators provide approximate timing information, however, none of these simulators provide the detailed timing information. Electrical circuit simulation is the only tool that provides accurate timing information and performance details. Typical timing accuracy is *1* nano-second and voltage tolerance is *0.001* Volts relative. This, however, requires more than three orders of magnitude more computing time compared to gate or switch level simulators [3].

The importance and the high computing cost of circuit simulation provides motivation for the development of fast and accurate electrical circuit simulators. These can be achieved by using improved algorithms and high performance computer architectures to run the simulation engines.

1.1 Circuit Simulation

Electrical circuit simulation programs such as SPICE [4], ASTP [5] and SLATE [6] are commonly used by IC designers. SPICE is the most popular circuit simulation program. Many different versions of SPICE such as SPICE2, and SPICE3 [7] and commercial implementations such as HSPICE, PSPICE, and IGSPICE are in use [8]. These simulators are robust and permit analysis of a wide variety of circuits. However, they are not cost-effective for the analysis of circuits with more than few hundred transistors.

Commonly used circuit simulators such as SPICE and HSPICE provide models for several nonlinear, active circuit devices such as field-effect transistors (FETs), bipolar-junction transistors (BJTs), and diodes. They offer a wide variety of analyses including dc analysis, time domain transient analysis, ac analysis, noise analysis, and distortion analysis. Among these, time domain transient analysis is the most expensive in terms of computer time. The present work focuses on techniques to improve the speed of time-domain transient analysis while maintaining acceptable waveform accuracy.

The transient analysis process can be divided into two stages: *equation formulation* and *equation solution*. Given the circuit external excitations and topological description, the equations are derived by means of Kirchoff's current law (KCL), Kirchoff's voltage law (KVL) and branch relations (BR). Branch relations are mathematical descriptions of the electrical behavior of circuit elements, for example the diode current-voltage relation. Applying KCL, KVL and BR to a circuit yields a set of nonlinear algebraic-differential equations (NL-ADEs). The unknowns are usually node voltages and branch currents. Two approaches are used to solve NL-ADEs. The first approach consists of transforming the NL-ADEs into nonlinear algebraic equations. Subsequently, a set of linear algebraic equations is derived. These equations are then solved using a conventional direct method, generally LU decomposition.

The second approach, developed more recently, employs iterative or relaxation, methods at different levels of the simulation process. Depending upon the stage of application of iterative techniques, relaxation techniques can be classified as *linear relaxation*, *non-linear relaxation* and *waveform*

relaxation [3]. Linear relaxation techniques are used to replace Gaussian elimination. Non-linear relaxation techniques such as non-linear Gauss-Seidel and Gauss-Jacobi techniques can be used to solve the system of non-linear algebraic equations. Waveform relaxation techniques use iterative techniques at the differential equation level. Simulators based on these algorithms provide waveforms as accurate as those of a standard circuit simulator, with up to two orders of magnitude speed improvement for large circuits [9,10,11]. These simulators have been used for simulation of both digital and analog MOS ICs. Relaxation methods lend themselves well to parallel processing and have been the subject of extensive research within the last few years [10, 11, 12, 13].

Conventional circuit simulators such as SPICE and ASTAP were designed for the analysis of circuits containing up to a few hundred transistors [14]. Many designers have used these programs for simulating circuits containing thousands of transistors even though the computation requirement is several CPU hours. Saleh and Newton [8] have reported that at some companies the SPICE program is executed over 50,000 times a month. It is observed that 80% are small jobs which consume 20% of the CPU time, while 20% of the jobs consume 80% of the CPU time.

The main reasons for high computing costs are briefly explained below. The time domain transient analysis involves formulation and iterative solution of nonlinear ordinary differential equations. The equation formulation is an $O(M)$ process where M is the total number of devices in the circuit. If M is large and device model equations are complex then the formulation process requires a large number of floating point operations. The order of complexity of each solution iteration varies from $O(N^{1.1})$ to $O(N^3)$, where N is the total number of equations in the system, depending on the sparseness of the system [3]. Several such iterations are required to obtain the solution of the system at each time point. Therefore, time-domain transient analysis is very expensive in terms of computer time. The importance and the CPU intensive nature of transient analysis is the motivation for the development of high performance simulators.

Two approaches have been used to improve the simulator performance. The first approach involves the development of new algorithms, such as

waveform relaxation and iterated timing analysis. These algorithms have proven effective for the analysis of MOS digital circuits. The second approach involves partitioning the circuit simulation problem into sub-problems such that a number of them can be evaluated concurrently using parallel processing techniques. More recently this approach has attracted a lot of attention. This research focuses on the application of parallel processing techniques to enhance performance of waveform relaxation techniques.

1.2 Parallel Processing

Parallel processing is a form of information processing that exploits concurrent manipulation of data elements to reduce total elapsed time to completion. Pipelining and task parallelism are two methods used to achieve concurrency [15]. Pipelining increases concurrency by dividing a computation into a number of steps. Each computation step is assigned to a pipeline stage. Buffers exist between pipeline stages and the pipeline control mechanism ensures that all the pipeline stages are evenly loaded. Parallelism is the use of multiple resources (processing elements). Parallel processor designs can be divided into three classes according to the number and complexity of the processing elements used: *fine-grain*, *medium-grain* and *large-grain* [16]. In fine-grain parallel processing systems, each processing element (PE) is typically capable of executing a few simple instructions, whereas each PE of a medium-grain system is able to process a procedure-sized group of instructions, and a PE of a large grain system may have the capacity of an entire modern day computer. Several medium-grain systems have been built in the last decade; their popularity can be partly attributed to the availability of off-the-shelf medium-grained components (microprocessors and memory units).

According to the number of instruction and data streams used, parallel systems can be classified as *single instruction, multiple data systems (SIMD)* and *multiple instruction, multiple data systems (MIMD)* [17]. SIMD operation involves multiple processors simultaneously executing the same instruction on different data. A wide variety of array processors fall in this category. MIMD operation involves multiple processors autonomously

executing diverse instructions on diverse data. Depending upon the inter processor communication technique used, MIMD systems can be further classified as *shared memory* and *message passing* [18]. Each of these has its advantages and disadvantages. The primary advantage of shared memory systems is the ability of parallel processes to share a single address space; a significant disadvantage is the bottleneck created by this shared resource. In addition, other important problems such as data access synchronization and cache coherence must be solved. In message passing architectures, PEs share data by passing messages. These architectures have been principally constructed in an effort to provide a parallel architecture that will scale (accommodate a significant increase in number of processors) well and will satisfy the performance requirements of large scientific applications characterized by local data references [16][19]. This architectural approach requires that a parallel program be divided into disjoint processes such that there is minimal communication between them. An important disadvantage of the message passing approach is the message latency as the data is queued and forwarded by intermediate PEs. It is important to note that neither of the two approaches described above is a clear cut winner. Substantial research is in progress to find a match between applications and suitable architectures.

As mentioned above, *medium grain* systems can be built using commercially available processors and memory chips. These are usually less expensive in comparison with *fine grain* and *large grain* systems. Hardware complexity and cost of medium grain message passing systems is usually lower than that of shared memory systems. Several medium grain systems are readily available in the market. Therefore, a medium-grained message passing architecture based on the *Inmos microprocessor T800* [20] is used for the project.

1.3 Motivation and Research Objectives

A number of approaches have been used to improve the performance of conventional DIRECT circuit simulators [3]. Look-up table techniques have been used to reduce the time required to evaluate complex device model equations [21, 22]. Techniques based on special purpose micro-code have

been investigated for reducing the time required to solve the sparse linear systems arising from the linearization of circuit equations [23]. Node tearing techniques have been used to exploit circuit regularity by bypassing the solution of sub-circuits whose state is not changing [24] and to exploit the vector processing capabilities of high performance computers such as the CRAY-1 [25]. In all the above cases, the overall speed improvement of the simulation has been at most an order of magnitude, for practical circuits [3].

Several commercial implementations of both shared memory and distributed memory parallel processing systems have become available in recent years. Multi-processor/multi-computer systems are attractive due to their low cost. These systems have been used to implement parallel direct method simulators. Parallel direct methods can exploit parallelism in formulation and solution of systems of equations. Newton and Sangiovanni-Vincentelli [3] have reported that for large circuits the majority of the time is spent in solution of linear systems of equations. The linear equation solution time grows faster than linearly with the circuit size. The LU factorization, forward elimination, and backward substitution used for the solution offers a limited amount of parallelism. In addition, the sparse and asymmetric nature of the system matrix makes parallel implementation difficult.

Relaxation techniques such as non-linear relaxation and waveform relaxation partition the system of equations into a number of sub-systems. Iterative techniques are used across sub-systems. This avoids parallel solution of large and sparse systems of linear equations. It is usually possible (depending on the algorithm used and the nature of the circuit) to solve sub-systems in parallel. Some parallel architectures also allow exploitation of fine grain parallelism within an iteration of a single sub-system. Thus parallelism available at various levels of the simulation process can be exploited. In addition, relaxation based simulators allow the use of waveform properties such as latency and multi-rate behavior to reduce the simulation time. Therefore parallel relaxation based simulators have become a focus of research. Two relaxation based techniques, parallel Iterated Timing Analysis and parallel waveform relaxation have been reported in the literature [11][10]. Waveform relaxation is a robust technique. It inherently exploits multi-rate properties. It is also possible to

reported in the literature [11][10]. Waveform relaxation is a robust technique. It inherently exploits multi-rate properties. It is also possible to organize parallel implementation in such a way that sub-systems exchange large relatively infrequent messages. This property is useful for parallel implementation on distributed memory machines.

Most of the relaxation based parallel simulator implementations described in the literature [10,11,12,13], with the exception of CONSISE (a simulator developed at CALTECH by Sven Mattison [12]), use shared memory multiprocessors. CONSISE does not perform block partitioning. Block partitioning and its impact on parallel implementation will be discussed later in the thesis. The message passing programming model is substantially different from a shared-memory programming model. Therefore detailed investigation of problems involved in implementing a circuit simulator on the more economical message passing systems is necessary.

While a few implementations have been described in the literature, there has been little done by way of performance analysis and evaluation of distributed memory parallel waveform relaxation. This is an area that deserves much more attention to identify bottlenecks of particular implementations or architectures.

The principal objective of this work is to study issues involved in the application of distributed memory parallel processing for the simulation of MOS digital VLSI circuits using waveform relaxation techniques. In particular, this thesis will determine how much the speed of calculation in circuit simulation can be increased with a low cost distributed memory parallel processing system. Also, a method will be sought to efficiently implement circuit simulation on the distributed memory machine.

The study involves analysis of different forms of parallelism in the direct method and the waveform relaxation technique. An analysis of various implementation strategies and their performance implications will be performed. A strategy appropriate for implementation on medium grained distributed memory machines will be selected. Two forms of the waveform relaxation algorithm will be implemented using a multi-transputer system to compare their performance characteristics. Load imbalance is a major

algorithm will be developed and implemented. Results of parallel implementation will be analyzed to identify sources of bottlenecks and possible remedies will be suggested.

1.4 Thesis Outline

This thesis describes important aspects of the distributed memory parallel waveform relaxation technique for the simulation of MOS VLSI circuits. The understanding of the research work requires background information in waveform relaxation techniques and parallel processing.

Chapter 2 reviews prior research on the waveform relaxation techniques. The chapter presents speed and robustness enhancement techniques, such as circuit partitioning and window selection. The waveform relaxation technique partitions a circuit into a number of sub-circuits. The direct method is used for simulation of sub-circuits, therefore, the direct method is also described. The topics presented in this chapter are mainly based on [3][14][8][26].

The objective of Chapter 3 is to describe issues and options involved in the parallel implementation of relaxation based circuit simulators. The information presented in this section can be divided into two logical parts. The first part consists of generic background on parallel processing issues. Important classes of parallel architectures are described and the effects of different forms of overheads on performance of parallel applications are analyzed. The second part is devoted to the analysis of issues involved in the parallel implementation of circuit simulation programs. Coarse and fine grain parallelism in the direct and relaxation methods are analyzed. The discussion in Section 3.3 and Section 3.4 is based on the work of Saleh et al. [27]. The discussion concentrates mainly on issues involved in parallel implementation of waveform relaxation programs on shared memory multi-processors. Issues specific to distributed memory machines such as partitioning and allocation are described in Sections 3.7 and 3.8.

Chapter 4 presents an implementation of a parallel waveform relaxation program. Two implementation strategies, a single queue and a distributed queue approach are compared. The parallel processing framework, and the

placement and partitioning algorithms used for the implementation are described.

An imbalanced workload in a parallel processing system results in low overall efficiency and speed-up. Load offered by a circuit simulator when simulating a large digital circuit changes with simulation time due to latency and multi-rate behavior. Therefore, load imbalance is an important source of overhead. Chapter 5 presents dynamic load balancing techniques used to reduce the load imbalance overhead. A implementation using a multi-transputer system is also presented.

Results of implementations described in Chapters 4 and 5 are given in Chapter 6. The effects of performance limiting factors are also analyzed. Finally Chapter 7 concludes the thesis, and gives suggestions for further research.

2. CIRCUIT SIMULATION TECHNIQUES

The last two decades have seen a substantial growth in the size and complexity of integrated circuits. Conventional prototyping techniques used to verify electronic circuit design and predict performance, such as breadboard implementation, are inadequate for large integrated circuits for several reasons. For example, modeling of parasitic components is difficult because the physical contexts of the prototype and the resultant system are so different. In addition, time and cost of prototype development is usually very high. Important goals of integrated circuit development are to minimize development time and to reduce risk. These goals fostered the development of computer programs to simulate integrated circuits. Early attempts in this direction led to the development of a simulation program CANCER [28]. SPICE, a successful circuit simulation program, evolved from CANCER. Later versions of SPICE, SPICE2 [4], and SPICE3 [7] added functionality and improved robustness. These programs use conventional Newton-Raphson based methods.

A wide variety of algorithms to improve the performance of simulators without sacrificing accuracy have been described in the literature. Of these approaches, the relaxation based approaches, such as, *waveform relaxation [9][10]*, *iterated timing analysis [11]* and *waveform-relaxation-Newton [11]* are ideally suited for simulating MOS digital circuits. This is due to the unidirectional nature of MOS devices. That is, due to the insulated gate terminal, the current through the gate is independent of the voltages at the other device terminals, if the effects of small gate-to-drain and gate-to-source capacitances are ignored [14]. The unidirectionality property helps relaxation decomposition, as will become clear later. This chapter describes basic mathematical techniques used for development of a circuit simulation program to perform time-domain transient analysis. Techniques used to formulate a system of non-linear ordinary differential equations are

described. Both direct and relaxation based techniques are discussed. Topics presented in this chapter are mainly based on [3][14][8][26][29][30].

2.1 Formulation Of The Equations

The first task performed by a circuit simulator is to read the circuit description and formulate a set of algebraic differential equations based on Kirchoff's Current (KCL) and Voltage law (KVL), and Branch relations. There are several different ways of formulating a system of equations. The most popular of these are Nodal Analysis, Modified Nodal Analysis and Sparse Tableau Analysis. Nodal analysis is the oldest and the most frequently used method [29]. Node voltages are unknown variables in this formulation. The main reason for popularity of nodal analysis is its simplicity. However it is difficult to simulate circuits with floating voltage sources and current controlled devices. Direct evaluation of branch currents is also difficult. Modified Nodal Analysis (MNA), implemented by Ho et al. [31], overcomes these difficulties. MNA can treat node voltages, voltage source currents, output currents and controlling source currents as unknown variables. The SPICE2 program uses MNA.

Sparse Tableau Analysis is the most general method. In this method, the set of unknown variables includes all branch voltages, branch currents and node voltages. This method formulates more equations per circuit than the previous two techniques, however the system of equations is usually very sparse and therefore the number of floating point operations necessary to solve these equations can be less than the number required to solve smaller, more dense systems such as those derived from nodal analysis [29].

MOS digital circuits seldom use floating voltage sources and current controlled devices. Therefore NA is an adequate formulation for their analysis. NA has been used in this thesis. The following assumptions are made while formulating the system of equations.

1. All resistive elements, including active devices, are characterized by constitutive equations where voltages are controlling variables and currents are controlled variables.

2. All energy storing elements are two-terminal, possibly nonlinear, voltage-controlled capacitors.

3. All independent voltage sources have one terminal connected to ground or can be transformed into independent current sources with the use of the Norton transformation.

Another important assumption required by relaxation-based simulators is that a two-terminal capacitor be connected from each node of the circuit to ground. This assumption is satisfied by circuits where lumped parasitic capacitances are present between circuit interconnect and ground or on the terminals of active circuit elements. This assumption helps in ensuring that the capacitance matrix has all non-zero diagonal elements. This point is further elaborated in Section 2.3.

The process of formulating a system of equations with node voltages as unknown variables consists of three steps. The first step is to use KCL to formulate a system of equations in terms of branch currents and node charges. The form of KCL which states that the algebraic sum of currents incident at a node must be equal to the rate of change of the algebraic sum of charges at the node, is assumed in the discussion. Then the branch currents and node charges are expressed in terms of branch voltages using branch relations. Finally, branch voltages are expressed in terms of node voltages using KVL.

The application of nodal analysis to a *(N+1)* node circuit yields *N* linear independent equations in *N* unknowns. Node *(N+1)* is treated as a reference or ground node and the corresponding equation is discarded. Thus, for each node in the circuit the following equation can be written:

$$\frac{d}{dt} \sum_{\text{capactive elements at } j} q_{element} + \sum_{\text{other elements at } j} i_{element} = 0. \tag{2.0}$$

The resulting system of *N* equations can be written in the form:

$$\dot{\mathbf{q}}(v(t), u(t)) = -\mathbf{f}(v(t), u(t)) \tag{2.1}$$

where $q(v(t), u(t)) \in \mathbf{R}^n$ is the vector of the sum of charges due to the capacitors connected to nodes, $\dot{\mathbf{q}}(v(t),u(t)) \in \mathbf{R}^n$ is the vector of time derivatives of $q(v(t), u(t)) \in \mathbf{R}^n$, $v(t) \in \mathbf{R}^n$ is the vector of node voltages at time t, $u(t) \in \mathbf{R}^n$ is the input voltage vector at time t and $f, f\colon \mathbf{R}^n \times \mathbf{R}^n \rightarrow \mathbf{R}^n$ is a vector function. It can be expressed as:

$$f(v(t),u(t)) = [f_1(v(t),u(t)), f_2(v(t),u(t)), \cdots, f_N(v(t),u(t))]^T$$

An ith element of f, $f_i(v(t),u(t))$, represents the sum of currents charging the capacitors connected to node i. Equation 2.1 is known as the charge formulation of the circuit equations because charge is treated as a state variable. It is also possible to treat voltage as the state variable. The resulting system of N equations can be written in the following form:

$$\begin{aligned} &C(v(t),u(t))\dot{v}(t) = -f(v(t),u(t)); \qquad t \in [0,T]; \\ &v(0) = V, \end{aligned} \tag{2.2}$$

where $\mathbf{C}(\bullet)\colon \mathbf{R}^n \rightarrow \mathbf{R}^{n \times n}$ represents the nodal capacitance matrix. The two formulations are equivalent for circuits with linear capacitors. However charge formulation must be used in circuits with nonlinear capacitors in order to keep the total charge in the system constant during the simulation process. Both formulations are used in this thesis. The equation formulation process can be explained with help of the following example [14]. Consider the nodal equation formulation for the MOS nand circuit of Figure 2.1:

The nodal equation for the first node is:

$$\begin{aligned} &i_{d_{m1}}(v_1, V_a, 0) - i_{d_{m2}}(v_2, V_b, v_1) + \frac{d}{dt}[q_{d_{m1}}(v_1, V_a, 0) + q_{s_{m2}}(v_2, V_b, v_1) + \\ &\quad C_1 v_1] = 0, \end{aligned} \tag{2.3}$$

and for the second node,

$$i_{d_{m2}}(v_2, V_b, v_1) + g_1 * (v_2 - V_{dd}) + \frac{d}{dt}[q_{d_{m2}}(v_2, V_b, v_1) + C_2 v_2] = 0, \tag{2.4}$$

where $i_{d_{m1}}$ and $i_{d_{m2}}$ are the currents from drains to sources of transistors M1 and M2 respectively, and $q_{d_{m1}}, q_{d_{m2}}, q_{s_{m2}}$ are the charges accumulated at the drain of transistor M1 and the drain and source of transistor M2, respectively. Although KVL equations have not been formulated explicitly, they have been used for expressing branch voltages in terms of node voltages. Current and charge terms in these equations can be expressed in terms of node voltages using branch relations. Branch relations are discussed in the following subsection.

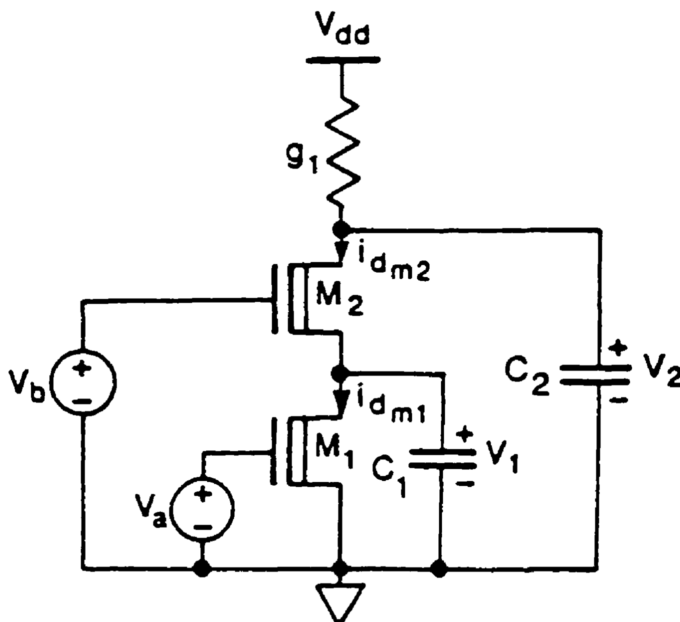


Figure 2.1: A MOS nand gate [14].

2.1.1 Branch Relations

Branch relations are mathematical descriptions of the electrical behavior of circuit elements. The branch relations can be divided into three different categories: resistive, capactive and inductive. Resistive and capactive branch relations are described because these are necessary for analysis of MOS digital circuits. Resistive branch relations relate voltages to currents and capacitive equations relate voltages to charges.

The branch relations for an n-terminal device can be represented by a set of *(n-1)* algebraic equations involving *(n-1)* terminal voltages and currents or

charges. One terminal is used as a reference and voltages of the other terminals are determined with respect to this reference terminal. The relation between diode current and anode-to-cathode voltage is an example of resistive branch relation. The current through the diode, i_a, can be computed from the following approximate equation:

$$i_a = I_s(e^{v_{ac}/V_t} - 1) \tag{2.5}$$

where v_{ac} is the anode-to-cathode voltage across the diode, I_s is the saturation current and V_t is the thermal voltage.

If the device currents can be uniquely determined from the equations, given the device voltages, then the device equations are said to be voltage-controlled. Often, given a set of device equations, it is possible to perform a transformation so that the device currents are explicit functions of device voltages. For example, Equation 2.5 is voltage-controlled because in it the current, i_a, is an explicit function of the device voltage, v_{ac}.

The commonly used approximate equations for a MOS transistor are another example of voltage-controlled device equations. The approximate device equations can be expressed as:

$$i_d = \frac{kW}{2L}[2(v_{gs} - v_T)v_{ds} - v_{ds}^2] \quad \text{for } v_{ds} \le v_{gs} - v_T; \tag{2.6}$$

$$i_d = \frac{kW}{2L}(v_{gs} - v_T)^2 \quad \text{for } v_{ds} \ge v_{gs} - v_T; \text{ and} \tag{2.7}$$

$$i_g = 0;$$

where i_d is the drain current, k is a parameter depending on the carrier mobility and thickness of the oxide, W and L are the width and the length of the channel of the transistor, v_{gs} is the gate-to-source voltage, v_{ds} is the drain-to-source voltage, v_T is the threshold voltage, and i_g is the gate current. The branch current equations for the MOS transistor are specified by two different algebraic functions, where the function is determined by the voltages at the terminals. Most of the devices in use today can be expressed by voltage controlled equations and therefore satisfy the first assumption mentioned above.

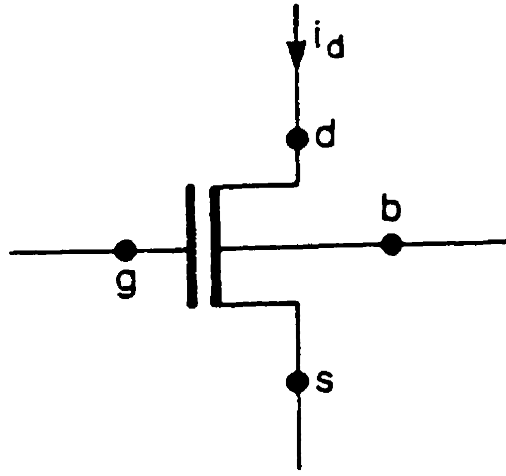


Figure 2.2: A MOS transistor in free space.

Branch equations for an n-terminal capacitive device are a set of *(n-1)* algebraic equations involving terminal voltages and terminal charges. For most commonly modeled devices, there exists a set of equations for the terminal charges that are voltage-controlled. For example, consider the junction capacitance for the diode for the case where the voltage across the diode $v_{ac} \leq 0.0$. Then, the anode charge, q_a, can be computed to the first order with the equation:

$$q_a = C_0 (\frac{1 - v_{ac}}{\phi})^{1-m} \tag{2.8}$$

where C_0 is the zero-bias junction capacitance and ϕ is the junction potential. Similarly, charge equations for capacitors of a MOS transistor (see Figure 2.2) in the saturation region are given by:

$$q_g = \frac{2}{3} WLC_{ox} (v_{gs} - v_T) + C_p (v_{gs} - v_{ds}); \text{ and} \tag{2.9}$$
$$q_d = 0;$$

where q_g and q_d are respectively the charge stored at the gate and the charge stored at the drain of the device, C_{ox} is the oxide capacitance, and C_p is small parasitic capacitance.

Given a physical device, its electrical behavior is best described by a combination of resistive and capacitive branch equations. Certain devices may require inductive branch equations as well. Inductive effects are usually considered for analysis of high frequency circuits. Analysis of MOS digital circuits seldom involves inductive branch equations. Therefore, these equations have been omitted from the discussion. Often the branch equations are symbolically represented by ideal elements such as two-terminal linear and nonlinear resistors, two terminal capacitors and controlled sources. For example, the MOS transistor model is represented by Figure 2.3.

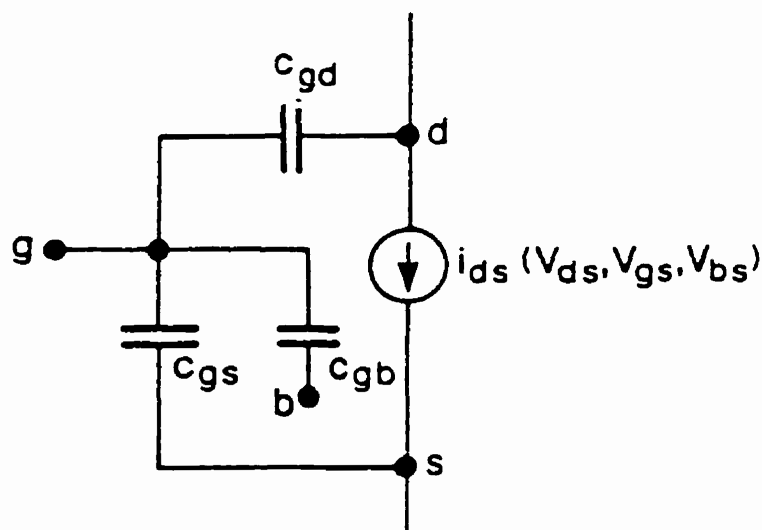


Figure 2.3: A MOS transistor model.

2.2 Direct Method

This section describes the procedure to obtain transient response of a given circuit using the direct method. As mentioned in the previous section, transient analysis involves formulation and solution of a system of nonlinear algebraic differential equations. Important mathematical steps involved in the simulation process are described below.

The first step is to formulate a system of equations similar in form to Equation (2.1) or Equation (2.2). The time derivative terms in the system of equations are then discretized using stiffly stable integration formulas, such as Backward Euler (BE), the Trapezoidal Rule (TR), or Gear's Variable-

Order Method (GE) [26][29]. An integration method divides the continuous simulation interval *[0, T]*, into a set of M discrete time points defined by

$$t_0 = 0, \qquad t_{n+1} = t_n + h_n, \qquad t_M = T.$$

The system of equations is solved at each time point t_{n+1}. The quantity h_n is known as the time step at time point t_n. The commonly used stiffly stable integration formulas are given below:

Backward Euler:

$$\dot{v}_{n+1} = \frac{v_{n+1} - v_n}{h_n}$$

Trapezoidal Rule:

$$\dot{v}_{n+1} = \frac{2}{h_n}(v_{n+1} - v_n) - \dot{v}_n$$

Gear's Variable Order Method:

$$\dot{v}_{n+1} = \frac{1}{h_n}\sum_{i=0}^{k} \alpha_i v_{n+1-i}$$

where v_n is the value of the unknown variable v at time t_n, h_n is the nth integration time step, and α and k are constants that depend on the order of the Gear's method. The application of integration formulas results in a set of nonlinear algebraic equations of the form:

$$g(v) = 0, \tag{2.10}$$

where $v \in \boldsymbol{R}^N$ is the vector of unknown voltages at t_{n+1}. These equations are then solved using a Newton-Raphson algorithm. A general form of the Newton-Raphson iteration equation to solve $F(v) = 0$, where $v \in R^N$ and $F{:}R^N \to R^N$ is:

$$J_F(v^k)(v^{k+1} - v^k) = -F(v^k), \tag{2.12}$$

where $J_F(v)$ is the Jacobian of *F(v)* and v^{k+1}, v^k are *k+1*th and *k*th iterates respectively. Thus the Newton-Raphson algorithm yields a set of linear equations of the form:

$$Av = b, \tag{2.5}$$

where $A \in R^{N \times N}$ is a matrix related to the Jacobian of g and $b \in R^N$. Typically less than 2 percent of the entries of A are non-zero for N > 500. The matrix is sparse because each node in the circuit is connected to only a few other nodes. These equations are then solved by using direct methods, such as sparse LU decomposition or Gaussian elimination. The Newton-Raphson process is iterated until convergence or until an upper bound on the number of iterations is reached. Typical bounds on convergence are 50 micro-volts absolute and 0.001 relative. A new time step is then selected. This procedure is continued until the simulation is complete.

Figure 2.4 shows a flowchart of the simulation process. The simulator operation begins by reading the circuit description. Values of the independent sources are computed at the present time point and values of the unknown variable are predicted using their values at previous time points. Each device is represented using resistors, capacitors and current sources. Integration formulas are applied to linear and nonlinear capacitors, nonlinear devices are linearised using the Newton-Raphson method and a system of linear equations is formulated. The process of applying integration formulas to capacitors and linearising nonlinear devices (steps 4, 5 and 6) is known as device model evaluation which is described in Appendix A. The system of equations is solved using sparse matrix techniques. Upon convergence of the NR iteration, local truncation error estimates are used to test the accuracy of the solution and to select a new time step. Time step control techniques are discussed in the following subsection. Steps (2) to (13) are repeated until the simulation is complete.

The direct method discussed above has proved to be reliable and accurate for simulation of a wide variety of circuits. However it has several limitations. Referring again to Figure 2.4, the majority of time spent in steps (2) to (13) can be lumped in two categories: the time required to form entries of A and b in Equation 2.5, the FORM phase (steps (5) and (6)), and the time required to solve the system of sparse linear equations, the SOLVE phase, (steps (7) and (8)). Several researchers have observed that for small circuits

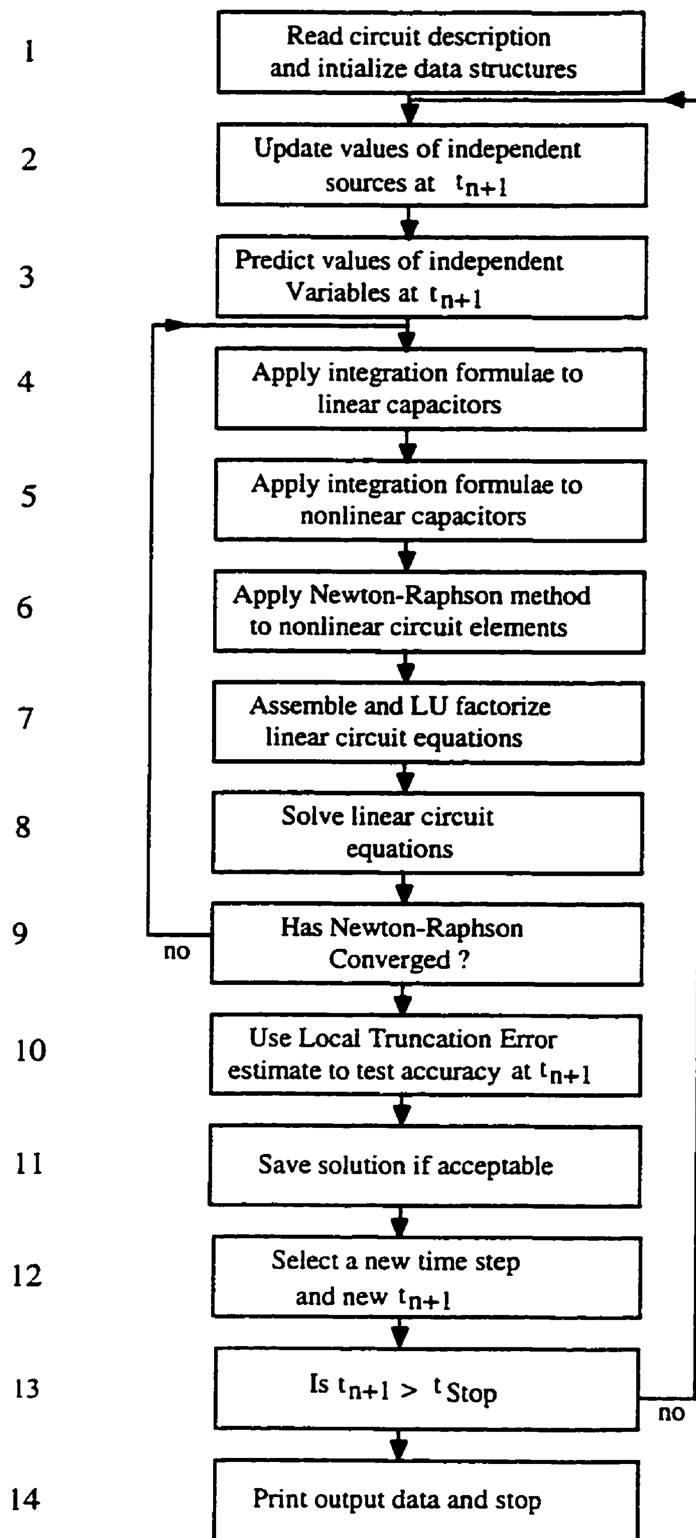


Figure 2.4: Flowchart of the simulation process [32].

(N < 20), the majority of the time is spent in performing the FORM, however, when the size of the circuit grows, an increasing percentage of the time is spent in the SOLVE phase [3]. The time spent in the SOLVE phase has been measured to grow as $O(N^{\beta})$ where $1.1 < \beta < 1.5$ [3]. On the other hand, the time required for FORM grows linearly with the number of circuit elements. Thus the SOLVE phase becomes a major bottleneck while simulating large circuits using the conventional approach. In addition, it is difficult to exploit waveform properties described later such as latency and multi-rate behavior. The relaxation-based techniques described in the next section attempt to solve these problems.

2.2.1 Time Step Control

The time required for simulating a circuit is proportional to the number of time steps necessary. Therefore, an objective of any circuit simulation algorithm is to reduce the number of time steps without sacrificing accuracy. Several schemes to select time steps for solving systems of NL-ADEs have been described in the literature. Commonly used time step control schemes for circuit simulation are: global fixed time step, iteration count, and Local Truncation Error (LTE) based time step control [4, 18, 26].

The fixed time step scheme selects the time step depending upon the fastest changing variable of a circuit. This scheme is very inefficient because several unnecessary time points are computed for time intervals when signals are changing slowly. The iteration count scheme uses some heuristic for selecting an initial time step. If the number of iterations required for convergence is larger than some N_{high}, the step size is reduced by some factor. If the number of iterations is less than N_{low}, the step size is increased by some factor. Otherwise the step size remains the same. This scheme is efficient; however there is no explicit accuracy control. It is commonly used in conjunction with the LTE-based scheme described below.

The LTE-based schemes observe the state of the circuit and the time step is adaptively changed accordingly. The Local Truncation Error (LTE) is the error made in one time step. If x_{n+1} is the numerical solution of a system of differential equations of the form

$$\dot{x}(t) = f(x(t), u(t)), \quad x(0) = X, \quad t \in [0, T], \tag{2.13}$$

at time point t_{n+1} and $x(t_{n+1})$ is the corresponding exact solution. Also if all previous solutions $(x_n, x_{n-1}, \cdots)$ are exact. Then the local truncation error for a general multistep integration method is defined as:

$$LTE_{n+1} = x(t_{n+1}) - x_{n+1}. \tag{2.14}$$

The local truncation error depends on the integration method and the time step. For example, the LTE for the trapezoidal method [26] can be shown to be

$$LTE_{n+1} = -\frac{h_n^3}{12}\frac{d^3x}{dt^3}(\xi) \qquad t_n \le \xi \le t_{n+1}. \tag{2.15}$$

The general form of the local truncation error [29] for most multistep integration methods of order k is given by

$$LTE_{n+1} = \tilde{C}_k h^{k+1} x^{(k+1)}(\xi) \qquad t_n \le \xi \le t_{n+1}. \tag{2.16}$$

where $\tilde{C}_k$ is a constant which depends on the integration method. The LTE-based time step control scheme estimates the LTE at each time step. The solution is accepted if the estimated LTE is less than the user-specified tolerance. The user-specified tolerance is usually expressed in terms of an absolute error parameter ε_a and a relative error parameter ε_r as

$$E_{UserLTE} = \varepsilon_a + \varepsilon_r \times \max|x_{n+1}, x_n| \tag{2.17}$$

The value of the derivate term $x^{(k+1)}(\xi)$ in the Equation 2.16 above is unknown. It can be approximated using divided difference as

$$x^{(k+1)}(\xi) \approx (k+1)!\,DD_{k+1}(t_{n+1}), \tag{2.18}$$

where $DD_{k+1}(t_{n+1})$ is the $k+1$st order divided difference. The LTE estimate is then

$$E_k = C_k h^{k+1} DD_{k+1}(t_{n+1}) \tag{2.19}$$

If predictor-corrector technique and BDF integration methods are used for the solution of differential equations then LTE can be easily estimated using the difference between the computed solution x_{n+1} and the predicted value $x^P(t_{n+1})$. The estimated LTE for a kth order BDF is given by [11]:

$$E_k = \left[\frac{h_k}{t_{n+1} - t_{n-k}}\right](x_{n+1} - x^P(t_{n+1})) \tag{2.20}$$

where the predicted solution $x^P(t_{n+1})$ can be computed using an explicit integration method. The predictor can be expressed as

$$x_{n+1}^0 = \sum_{i=1}^{k+1} \gamma_i x_{n+1-i} \tag{2.21}$$

where the superscript *0* indicates the *0*th iteration and the γ_i values are selected such that the predictor x_{n+1}^0 is correct if the solution is a *k*th order polynomial. Usually a *k*th-order predictor is used with a *k*th order integration method. The computed solution x_{n+1} is accepted if

$$|E_K| < E_{UserLTE}.$$

Saleh [11] has proposed a convenient method of implementing this check which is described below. A ratio r of allowable LTE and the actual LTE is computed.

$$r = \left|\frac{E_{UserLTE}}{E_k}\right| = \frac{|C_k h_{\max}^{k+1} \ddot{x}(\xi)|}{|C_k h_n^{k+1} \ddot{x}(\xi)|} \tag{2.22}$$

Therefore

$$r = \left[\frac{h_{\max}}{h_n}\right]^{k+1} \tag{2.23}$$

and

$$r_{LTE} = \left[\frac{h_{\max}}{h_n}\right] = (r)^{(\frac{1}{k+1})} \tag{2.24}$$

If

$$r_{LTE} > 1.0,$$

then the solution is accepted. The ratio can be also used to compute the next time step. The next recommended step is given by

$$h_{n+1} = r_{LTE} h_n \tag{2.25}$$

In addition to limits imposed by the local truncation error, a few practical considerations are also used for selecting the next time step. For example, time step is selected in such a way that steps fall on input break points and window boundaries. Bryton et al. [33] have observed that in several practical cases, rapid changes in step sizes introduce instability problems. Therefore it is necessary to limit changes in step size. The Relax2.3 and iSPLICE3 [14, 8] programs use four parameters, s_l, s_u, α , and β to control changes in step size. The step can be reduced at most by a factor s_l and increased by a factor s_u,. The factor α permits the same step size to be used a number of times and β is a growth factor. The strategy used for limiting changes in step size is described below. If $r_{LTE} < 1.0$ then the step size is reduced by a factor $MAX(s_l, r_{LTE})$. If $1.0 < r_{LTE} < \alpha$, the same step size is maintained. Similarly if $r_{LTE} \geq \alpha$ then the step size is increased by $MIN(s_u, \beta r_{LTE})$. Typical values of the parameters are $s_l = 0.25$, $s_u = 2.0$, $\alpha = 1.2$ and $\beta = 0.9$.

2.3 Relaxation Methods

Relaxation methods are numerical techniques used for the solution of a system of linear, nonlinear and differential equations. The basic structure of a relaxation-based simulator is similar to the standard circuit simulator. A set of NL-ADEs, in the form of Equation 2.2 or Equation 2.3, are formulated using KCL, KVL and branch relations and relaxation-based techniques are used to solve them. The relaxation techniques have two advantages: they do not require direct solution of a large system of linear equations, resulting in a considerable reduction in the SOLVE bottleneck, and they permit the simulator to exploit latency and multi-rate behavior efficiently [11]. Latency and multi-rate behavior are discussed below.

As discussed in Section 2.2, most direct method circuit simulators use a common time step for the complete circuit. This results in computation to solve for each variable at every time point. The time step at a time point is computed by calculating a minimum of recommended time steps for all variables. Therefore the fastest changing variable in the system determines the time step. As a result, several extra points are calculated for slowly changing variables than necessary to represent the variable accurately. This effect is particularly significant for variables that are not changing

appreciably over some interval of time. Waveform latency refers to the situation where a variable is not changing appreciably over some interval of time and its solution can be obtained from the explicit equation:

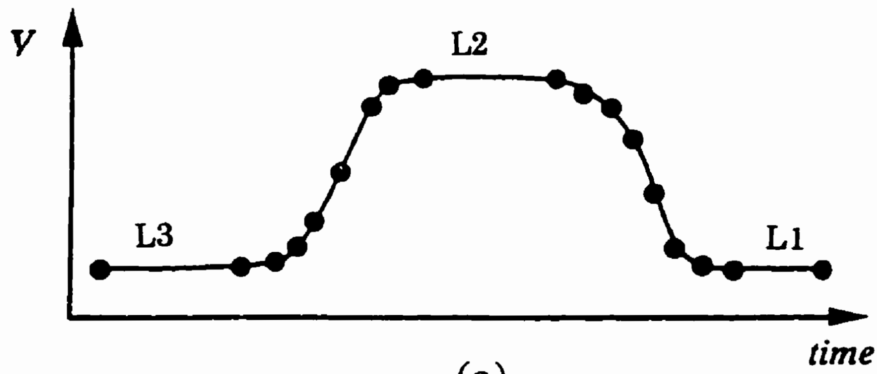
$$x_{n+1} = x_n \ .$$

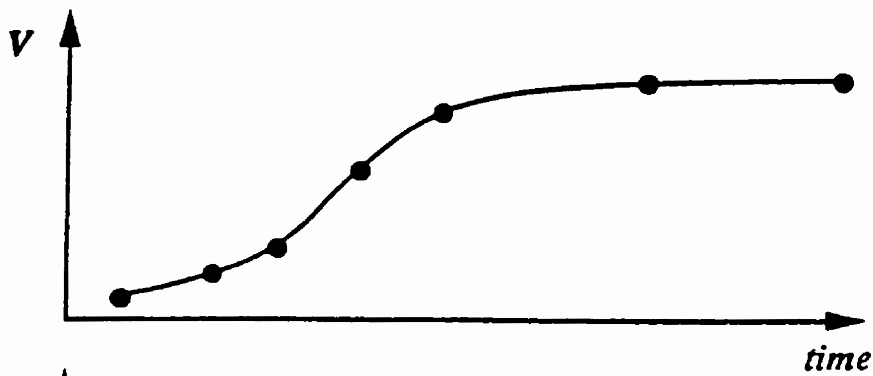
That is, the value x_{n+1} is not computed using a numerical integration formula but instead is updated using the value at the previous time point. Latency is a well-known property of large digital circuits. Figure 2.5a shows a node voltage waveform with three latent regions, *L1*, *L2* and *L3*, where node voltages are updated using their previous values. Multirate behavior refers to signal values changing at different rates, relative to one another, over the same interval of time. MOS digital circuits show multirate behavior because of different transistor sizes and capacitance values. Figure 2.5b shows voltage waveforms of two nodes of a sub-circuit which are changing at different rates where large time steps are used to obtain the slowly changing, first, waveform and smaller time steps are taken to obtain the rapidly changing, second, waveform. This is an example of the multirate behavior. An effective exploitation of latency and multirate behavior can result in significant reduction in simulation time.

As shown in Figure 2.6, relaxation based techniques can be used at various stages of the solution process. Depending upon the stage at which the relaxation process is applied, the techniques are classified as *Linear, Nonlinear* or *Waveform*. The most commonly used numerical analysis techniques for the purpose are Gauss-Seidel and Gauss-Jacobi.

2.3.1 Linear Relaxation

Linear relaxation techniques use iterative methods for the solution of system of linear equations. Referring to the flow chart of a direct method simulator shown in Figure 2.4, either Gauss-Seidel or Gauss-Jacobi methods can be used instead of LU decomposition or Gaussian elimation in steps (7) and (8). GS and GJ algorithms used for the solution of equation $Ax = b$, are given below. The constant ε indicates the error tolerance and k is the iteration count.





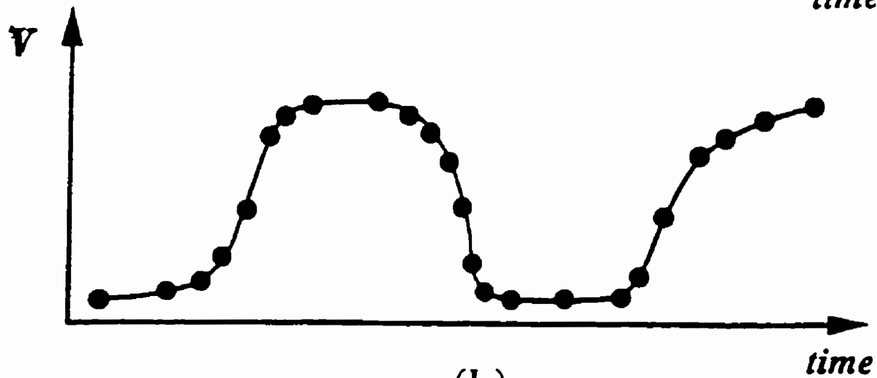


Figure 2.5: Waveform properties [a] Latency [b] Multi-rate behavior [8].

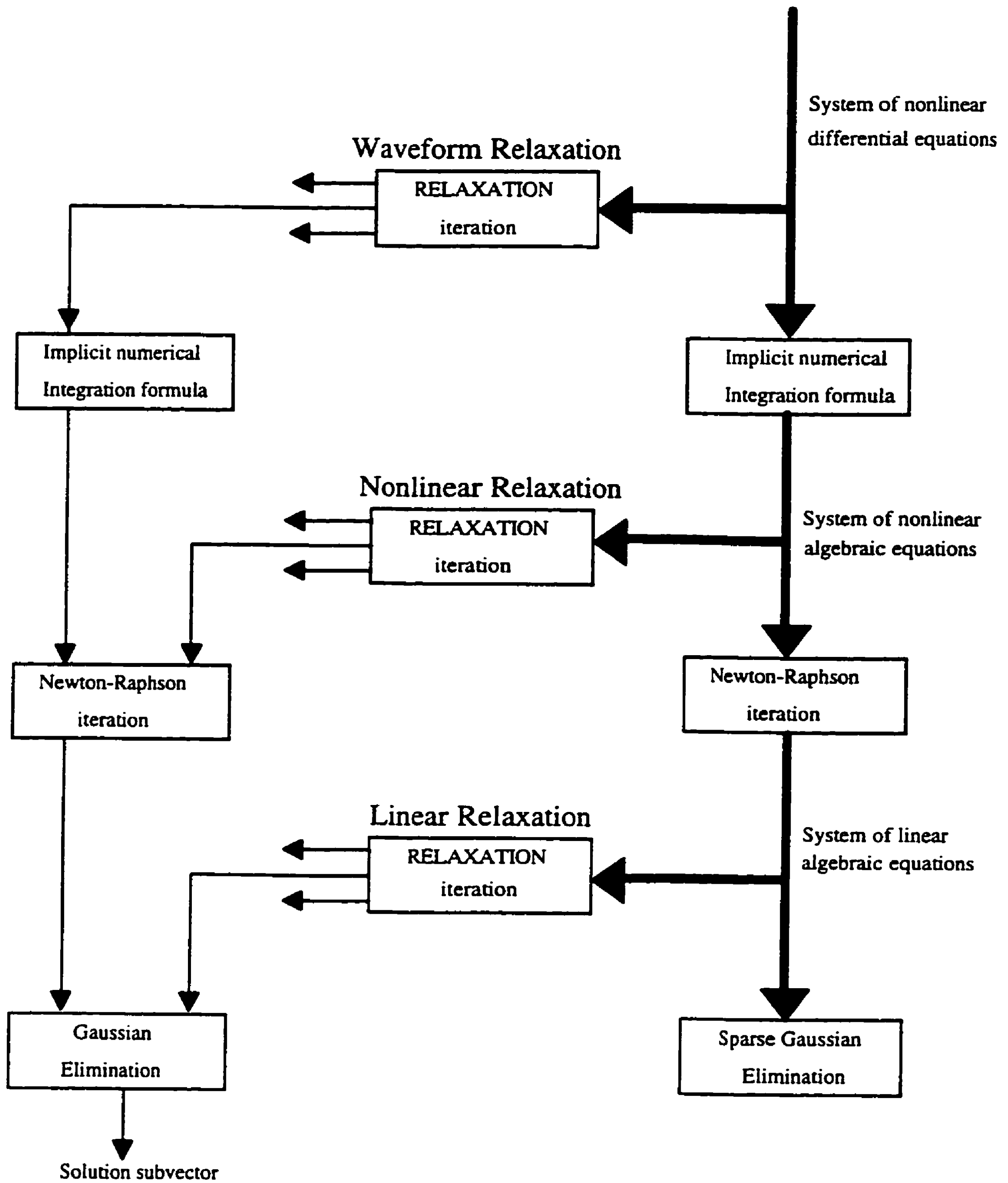


Figure 2.6: Relaxation-based techniques [9].

Gauss-Jacobi Algorithm to Solve $Ax = b$

$k \leftarrow 0$
estimate x^0;
repeat {
    $k \leftarrow k+1$;
    forall ($i \in \{1,\dots,n\}$)

$$x_i^k = \frac{1}{a_{ii}}\left[\quad b_i - \sum_{j=1}^{i-1} a_{ij}x_j^{k-1} - \sum_{j=i+1}^{n} a_{ij}x_j^{k-1}\quad\right]$$

} **until** ($|x_i^k - x_i^{k-1}| \le \varepsilon, \quad (i = 1,\cdots,n)$);

Gauss-Seidel Algorithm to solve $Ax = b$

$k \leftarrow 0$
estimate x^0;
repeat {
    $k \leftarrow k+1$;
    foreach ($i \in \{1,\dots,n\}$)

$$x_i^k = \frac{1}{a_{ii}}\left[\quad b_i - \sum_{j=1}^{i-1} a_{ij}x_j^{k} - \sum_{j=i+1}^{n} a_{ij}x_j^{k-1}\quad\right]$$

} **until** ($|x_i^k - x_i^{k-1}| \le \varepsilon, \quad (i = 1,\cdots,n)$);

It is important to note that in the GJ method, each iteration x_i^k is calculated by using values from the previous iteration, x_j^{k-1}. Therefore equations can be solved in any order or can be scheduled in parallel on multiple processors. In the GS method, the most recent (i.e. latest k) iteration values are used for calculating the current iteration. Therefore the order of processing equations is important. This is indicated by the use of the *foreach* construct.

In order to study convergence properties of the GS and GJ methods, A in the equation $Ax = b$ can be expressed as $A = L+D+U$, where $L \in R^n$ is strictly

lower triangular, $D \in R^n$ is diagonal and $U \in R^n$ is strictly upper triangular. The application of GJ and GS methods results in the following equations:

Gauss-Jacobi:

$$x^{k+1} = -D^{-1}(L+U)x^k - D^{-1}b = M_{GJ}x^k + D^{-1}b. \qquad (2.26)$$

Gauss-Seidel:

$$x^{k+1} = -(L+D)^{-1}Ux^k + (L+D)^{-1}b = M_{GS}x^k + (L+D)^{-1}b \qquad (2.27)$$

A necessary and sufficient condition for iterations defined by Equations 2.26 and 2.27 to converge to the solution of equation Ax = b, independent of any starting vector x^0, is that the eigenvalues of M_{GS} and M_{GJ} be inside the unit circle in the complex plane [34]. If these conditions are satisfied then GS and GJ methods converge at least linearly. That is, the error at each iteration decreases according

$$\|x^{k+1} - \hat{x}\| \quad \le \quad \varepsilon\|x^k - \hat{x}\| \qquad (2.28)$$

where $\hat{x}$ is the solution of the equation $Ax = b$.

The two relaxation based methods can be compared with the standard approach. The direct methods are certainly more reliable, however, their computational cost is $N^{1.1-1.5}$, compared with *O(N)* for the relaxation based methods [3]. Thus relaxation methods are advantageous from a computational point of view, if the number of iterations required to converge is of the order of $N^{0.1}$.

It is also possible to compare the Gauss-Seidel approach with the Gauss-Jacobi approach. Gauss-Seidel can be shown to converge faster than Gauss-Jacobi in most cases. For example, if A is lower triangular, Gauss-Seidel can converge to the solution of the system of equations in one iteration while Gauss-Jacobi converges in N iterations. However, in the Gauss-Jacobi approach, computations for all the equations can proceed in parallel,

therefore this technique is appropriate for parallel systems having a large number of processors.

2.3.2 Nonlinear Relaxation Methods

As mentioned in the earlier section, the first step in solving a system of differential equations obtained from the circuit description is to convert it to a system of nonlinear algebraic equations using integration formulas (refer Equation 2.4). The system of nonlinear equations thus formed can be solved using relaxation techniques without linearising them. This technique is known as nonlinear relaxation.

Consider a system of nonlinear equations F(x) = 0, $F: R^n \rightarrow R^n$ with components $(f_1, f_2, \cdots, f_n)$ and $f_i: R^n \rightarrow R$. Gauss-Seidel and Gauss-Jacobi algorithms for solution of *F(x) = 0* are given below. The index *k* is the iteration count and $\varepsilon_1, \varepsilon_2$ are error tolerances

Nonlinear Gauss-Seidel Algorithm:

$k \leftarrow 0;$

assume x^0

repeat{

$\quad k \leftarrow k+1;$

$\quad$ *foreach* $(i \in \{1, \cdots, n\})$

$\quad\quad$ *solve* $f_i(x_1^k, \cdots, x_{i-1}^k, x_i^k, x_{i+1}^{k-1}, \cdots, x_n^{k-1}) = 0$ *for* x_i^k;

} ***until***$(|x_i^k - x_i^{k-1}| \leq \varepsilon_1, |f_i(x^{k,i})| \leq \varepsilon_2, \quad i = 1, \cdots, n)$

The *foreach* construct specifies that the computations for each value of *i* must proceed sequentially and in the specified order. As mentioned earlier, convergence of this method depends on the order of processing equations. The order can be determined statically or dynamically.

Nonlinear Gauss-Jacobi Algorithm:

$k \leftarrow 0;$

assume x^0

repeat {

$\quad k \leftarrow k+1;$

$\quad$ *forall* $(i \in \{1,\cdots,n\})$

$\quad\quad$ *solve* $f_i(x_1^{k-1},\cdots,x_{i-1}^{k-1},x_i^k,x_{i+1}^{k-1},\cdots,x_n^{k-1}) = 0$ *for* x_i^k;

} ***until*** $(|x_i^k - x_i^{k-1}| \le \varepsilon_1, |f_i(x^{k,i})| \le \varepsilon_2, \quad i = 1,\cdots,n)$

The *forall* construct specifies that the computations for all values of i can proceed concurrently, i.e. in parallel and in any order.

The conditions under which these methods converge are similar to the linear case. Let $F'(x)$ denote the Jacobian of F at x and let $\hat{x}$ be the solution vector, i.e. $F(\hat{x}) = 0$. Assume that F is continuously differentiable in the open neighborhood S_0 of $\hat{x}$. The Jacobian $F'(x)$ can be expressed as $L(\hat{x}) + D(\hat{x}) + U(\hat{x})$ where $L(\hat{x})$, $D(\hat{x})$, and $U(\hat{x})$ are lower triangular, diagonal and upper triangular parts respectively. Let $M_{GJ}(\hat{x})$ and $M_{GS}(\hat{x})$ be defined as:

$$M_{GJ}(\hat{x}) = -D(\hat{x})^{-1}(L(\hat{x}) + U(\hat{x})), \tag{2.29}$$

$$M_{GS}(\hat{x}) = -(D(\hat{x}) + L(\hat{x}))^{-1}U(\hat{x}). \tag{2.30}$$

If $D(\hat{x})$ is non-singular and the spectral radii $\rho(M_{GS}(\hat{x})) < 1$, $\rho(M_{GJ}(\hat{x})) < 1$, then there exists an open ball $S \subset S_0$ such that nonlinear Gauss-Jacobi and Gauss-Seidel methods converge to the solution $\hat{x}$ for any starting vector $x^0 \in S$ [35]. It is important to note that unlike linear relaxation, convergence is guaranteed only if the initial guess (starting vector) is sufficiently close to a solution.

The nonlinear Gauss-Seidel and Gauss-Jacobi algorithms presented above assume that the equations can be solved exactly. However, since these are nonlinear equations, iterative methods must be used for obtaining their solution. Each equation, in one unknown, is usually solved using a single variable Newton-Raphson method. The resulting composite methods are known as Gauss-Seidel-Newton and Gauss-Jacobi-Newton. Ideally the inner Newton-Raphson loop should be iterated to convergence. However, it is found that a single iteration of the loop is usually adequate [35].

Nonlinear iteration methods can be compared with the direct Newton-Raphson method. It is observed that the convergence rate of direct Newton-Raphson methods is quadratic while it is only linear for relaxation based methods. However, each iteration of the direct Newton-Raphson method requires solution of a set of simultaneous equations while the relaxation method involves a set of decoupled equations. Thus, relaxation methods have better inherent parallelism. In addition, relaxation methods are ideally suited to exploit the latency of the circuit under analysis.

A comparison can be made of the use of relaxation methods at the linear and nonlinear equation level. The linear relaxation methods have an outer Newton-Raphson loop and an inner relaxation loop embedded in it, while nonlinear relaxation methods have an outer relaxation loop and an inner Newton-Raphson loop, operating independently on each equation, embedded in it. The use of relaxation at the linear equation level involves computation of the Jacobian of F, which is quite expensive. Nonlinear relaxation coupled with an inner Newton-Raphson only needs the partial derivative of f_i, with respect to x_i, resulting in a considerable saving of computer time per iteration.

The application of iterative methods at the linear and nonlinear equation level has been presented. It is also possible to use these techniques at the differential equation level. This method is known as Waveform Relaxation and is discussed in Section 2.5. The following section describes a circuit simulation approach based on nonlinear relaxation, known as Iterated Timing Analysis (ITA).

2.4 Iterated Timing Analysis

Iterated Timing Analysis is an electrical circuit simulation approach suitable for simulation of large MOS digital circuits. Enhancements in the timing analysis algorithms led to the development of ITA. Nonlinear relaxation based timing simulators such as MOTIS [22] and SPLICE1 [3], solve each nodal equation by performing only one relaxation iteration and one or more Newton-Raphson iterations per time step. It is assumed the correct solution can be obtained by appropriate selection of the time step. Performing only one relaxation iteration results in a substantial reduction in simulation time, however it is impossible to guarantee accuracy for an arbitrary connection of MOSFETs. In addition, it is difficult to simulate circuits with tight feedback loops and floating capacitors. Therefore timing simulators have not been successful for simulation of custom VLSI circuits. ITA based simulators overcome these difficulties by continuing the relaxation process to convergence at each time point. The circuit simulator, iSPLICE3 [8], developed at the University of Illinois at Urbana-Champagne is based on ITA. It has been used for simulation of large MOS digital and complex analog circuits. ITA is also amiable for implementation on advanced computer architectures such as vector and array processors as well as data-flow machines.

Important steps involved in ITA are described below. A system of nonlinear ordinary differential equations similar in form to Equation 2.1 or 2.2 is formulated. It is discretized and converted to a system of nonlinear algebraic equations using a stiffly stable integration formula. The resulting system of NL-ADEs is then solved using nonlinear Gauss-Seidel or Gauss-Jacobi method. Only one iteration of the inner Newton-Raphson loop is performed.

A special feature of ITA is the use of an event driven selective trace technique for exploiting latency. In the selective trace technique, every circuit node maintains two tables; one contains the names of its fan-in elements and the other contains the names of fan-out elements. Whenever the voltage at a node changes, all its fan-out elements are scheduled for processing. In this way, the effect of a change at the input to a circuit may be

traced as it propagates to other circuit nodes via the fan-out tables and the circuit elements which are connected to them. Only nodes directly affected by the change are processed. Therefore this technique is selective. It is more efficient than the bypass techniques used with standard circuit simulators [6].

ITA uses nonlinear relaxation techniques, therefore its convergence properties are identical to the basic nonlinear relaxation, as discussed in greater detail below. It was shown in the previous section that convergence requires diagonal dominance of the Jacobian of the discretized nonlinear equations. Referring again to Equation 2.2 ,

$$C(v,u)\dot{v} + f(u,v) = 0, \qquad v(0) = V$$

where C is the capacitance matrix in which, C_{ij}; $i \neq j$, is the total floating capacitance between nodes i and j, and C_{ii} is the sum of the capacitances of all capacitors connected to node i. f is a continuous function, each component of which represents net current charging the capacitor at a node due to other conductive elements. Now, if $C(v,u)$ is assumed to be symmetric and positive definite, and hence strictly diagonally dominant, then it can be shown that the Jacobian matrix of the discretized nonlinear circuit equations is also diagonally dominant provided that the time step is small [3]. As mentioned above, the diagonal dominance of the Jacobian is necessary to guarantee convergence of relaxation-based methods. Thus, diagonal dominance of the capacitance matrix can be used to check the diagonal dominance of the Jacobian matrix. The assumption made about the capacitance matrix holds if all the capacitors in the circuit are two terminal and are positive for all values of v. Lelarasmee [9] has proved the convergence properties of ITA.

The important steps involved in Gauss-Seidel ITA are given in the following algorithm; the algorithm can be easily modified to GJ form. For every simulation time point, two event lists, $E_A(t_n)$ and $E_B(t_n)$ are generated. These are used to separate the nodes that are to be processed in the successive iterations, k, $k+1$, of the Gauss-Seidel-Newton process.

Gauss-Seidel ITA Algorithm:

Put all nodes that are connected to independent sources in event list $E_A(0)$:

$t_n = 0$;
while ($t_n < TSTOP$)

  $k \leftarrow 0$;
  while (event list $E_A(t_n)$ is not empty){
    foreach (i **in** $E_A(t_n)$){
      obtain v_i^{k+1} from $g_i(v_1^{k+1}, \cdots, v_i^{k+1}, \cdots, v_N^k) = 0$
      using single Newton-Raphson step

      if ($|v_i^{k+1} - v_i^k| \leq \varepsilon$; i.e. convergence is achieved){
        use LTE to determine the next time t_s for processing node i;
        add node i to event list $E_A(t_s)$;
      }
      else {
        add node i to event list $E_B(t_n)$;
        add the fanout nodes of node i to event list $E_A(t_n)$;
      }
    }
    $E_A(t_n) \leftarrow E_B(t_n)$; $E_B(t_n) \leftarrow empty$;
    $k \leftarrow k + 1$;
  }
  $t_n \leftarrow t_{n+1}$;
}

where t_n is the present time for processing and t_{n+1} is the next time at which an event will be scheduled. Detailed discussion on the time step control technique is given in Deutsch [13]. Several variants of the ITA technique and latency exploitation schemes are discussed in [11].

2.5 Waveform Relaxation

The previous two sections demonstrate the application of relaxation techniques at the linear and nonlinear equation level. As mentioned earlier, relaxation techniques can also be applied at the differential equation level.

This technique is known as waveform relaxation (WR). Waveform relaxation has been derived from Picard iteration and classical relaxation techniques such as Gauss-Seidel and Gauss-Jacobi. In waveform relaxation, iteration variables are node voltage waveforms. This section reviews the basic WR technique and presents a few extensions. Waveform relaxation can be best explained with the help of the following example.

Consider a system of equations in $v(t) \in \boldsymbol{R}^2$ on $t \in [0,T]$

$$\dot{v}_1 = f_1(v_1, v_2, t) \qquad v_1(0) = v_{10} \tag{2.31}$$

$$\dot{v}_2 = f_2(v_1, v_2, t) \qquad v_2(0) = v_{20} \tag{2.32}$$

The Gauss-Seidel and Gauss-Jacobi forms of waveform relaxation are defined by the iterations:

Gauss-Jacobi WR:

$$\dot{v}_1^i = f_1(v_1^i, v_2^{i-1}, t) \tag{2.33}$$

$$\dot{v}_2^i = f_2(v_2^i, v_1^{i-1}, t) \tag{2.34}$$

Gauss-Seidel WR:

$$\dot{v}_1^i = f_1(v_1^i, v_2^{i-1}, t) \tag{2.35}$$

$$\dot{v}_2^i = f_2(v_2^i, v_1^i, t) \tag{2.36}$$

The basic idea of Gauss-Seidel waveform relaxation is to assume an initial value for the waveform $v_2:[0,T] \to \boldsymbol{R}$ and solve Equation (2.31) as an equation in one variable v_1 on the time interval $[0,T]$. The waveform thus obtained for v_1 is substituted in Equation (2.32) making it an equation in one variable v_2. The waveform for v_2 obtained by solving Equation 2.32 is then substituted in Equation 2.31. This process is iterated until waveforms for v_1 and v_2 converge. In Gauss-Seidel waveform relaxation, order of solution of equations is important, as waveforms obtained during the current iteration

are used to solve subsequent equations. In Gauss-Jacobi WR, waveforms obtained from the previous iteration are used to solve all the equations. Therefore equations can be solved in any order. Thus waveform relaxation replaces the problem of solving a system of differential equations in two variables by one of solving a sequence of differential equations in one variable. So it is a technique for time-domain decoupling of differential equations. It is important to note that each equation is solved at its own rate using an independent time step control mechanism. This allows the use of small time steps for rapidly changing variables and large time steps for slowly changing variables.

A Gauss-Seidel form of the WR algorithm for transient analysis of an electrical circuit is described below. Recall the system of NL-ADEs used to describe an electrical circuit,

$$C(u,v)\dot{v} + f(u,v) = 0, \qquad v(0) = V,$$

where $C(u,v)$ is a capacitance matrix and f is a continuous function each component of which represents the net current charging the capacitor at each node.

The convergence of the WR method is guaranteed under the following conditions. If $C(v(t),u(t)) \in R^{n \times n}$ is strictly diagonally dominant uniformly over all $v(t) \in R^n$ and $u(t) \in R^r$ and Lipscitz continuous with respect to $v(t)$ for all $u(t)$, then the sequence of waveforms generated by the Gauss-Seidel or Gauss-Jacobi WR algorithm will converge to the solution of Equation 2.2 independent of the initial guess [10].

In terms of the equation above, the Guess-Seidel algorithm can be written as follows [8]:

Gauss-Seidel WR Algorithm:

$k \leftarrow 0$

guess waveform $v^0(t);\ t \in [0,T]$ *such that* $v^0(0) = v_0$

(*for example, set* $v^0(t) = V, t \in [0,T]$);

repeat

{

$k \leftarrow k+1;$

foreach $(\ i \in (1,\cdots,n)\ )$

{

solve

$$\sum_{j=1}^{i} C_{ij}(v_1^k,\cdots,v_i^k,v_{i+1}^{k-1},\cdots,u)\dot{v}_j^k +$$

$$\sum_{j=i+1}^{n} C_{ij}(v_1^k,\cdots,v_i^k,v_{i+1}^{k-1},\cdots,u)\dot{v}_j^{k-1} +$$

$$f_i(v_1^k,\cdots,v_i^k,v_{i+1}^{k-1},\cdots,u) \quad = \quad 0$$

for $(v_i^k(t);\quad t \in [0,T])$*, with the initial condition* $v_i^k(0) = V_i.$

}

}

until $(max_{1 \le i \le n} max_{[t] \in [0,T]} |v_i^k(t) - v_i^{k-1}| \le \epsilon)$

A Gauss-Jacobi form of the WR algorithm can be obtained from the GS WR algorithm by replacing the ***foreach*** statement by a ***forall*** statement and adjusting the iteration indices in such a way that strictly previous iteration values are used for calculating v_i^k. The algorithm shown above partitions a given system of equations into a number of sub-systems each consisting of one unknown variable. This is equivalent to partitioning a given circuit into a number of sub-circuits each consisting of one electrical circuit node. A description of WR in terms of an electrical circuit is given below.

A circuit is partitioned into a number of sub-circuits. Each sub-circuit is simulated using an implicit integration and the Newton-Raphson algorithm at its own rate. Output waveforms of a sub-circuit are used as inputs for its

fanout sub-circuits. The sub-circuit simulation process is iterated until all node waveforms converge within a user-specified tolerance. In Gauss-Seidel WR, present iteration waveforms are used for simulating fanout sub-circuits. Therefore sub-circuits are ordered in accordance with the direction of signal flow. In the Gauss-Jacobi algorithm, previous iteration waveforms are used for simulating fanout sub-circuits. Therefore sub-circuits can be simulated in any order.

The gain of the WR approach is approximately determined by the following quotient:

$$G_{WR} \cong \frac{(\text{Matrix Partitioning})(\text{Multirate Factor})}{(\text{No. WR Iterations})(\text{Implementation Factor})} [37].$$

The GS WR algorithm shown above decomposes a given system of equations into a number of systems each having strictly one unknown variable. It is also possible to decompose the system of equations in such a way that each sub-system contains more than one unknown. In this case, the direct method is used for solving sub-systems and an iterative technique is used across sub-systems. This technique is discussed in the following section. The *Matrix Partitioning* yields an improvement due to the solution of smaller matrices instead of a large sparse matrix. The *Multirate Factor* also results in substantial gains for large circuits. Large digital circuits usually have 0.01 to 20 percent of a circuit active at a typical point. In addition circuit partitioning allows effective exploitation of multirate behavior since multirate activities can be contained in the sub-circuits.

The gain of the WR approach is inversely proportional to the number of waveform iterations. Techniques for reducing the number of WR iterations are discussed in the following two sub-sections. An average of *2* to *4* WR iterations is typical for a simple circuit. The efficiency of the programming implementation can have significant impact on performance. Therefore appropriate selection of data structures and algorithms is important. This effect is represented by the implementation factor.

2.5.1 Windowing Mechanism

The conditions for convergence mentioned above are satisfied by a wide variety of combinational, sequential and analog circuits. However, it is observed that circuits with tight feedback loops require large numbers of iterations for convergence. In addition, the number of iterations required to converge is proportional to the simulation interval [14]. This motivated development of a windowing mechanism. In this scheme, the simulation interval *[0, T]* is divided into a number of sub-intervals $[0, T_1], [T_1, T_2], [T_2, T_3], \cdots, [T_{n-1}, T]$, known as windows. Waveform relaxation is used for computing waveforms for the first window and the values of node voltages at T_1 are used as initial conditions for the analysis of the second window; this process is repeated for the analysis of all windows. An appropriate selection of window size is important for minimizing the time required for simulating a circuit. WR converges more rapidly as window size is reduced. However as window size is reduced, some advantages of WR are lost. Very small windows limit time steps selected to compute a waveform resulting in unnecessary calculations. A large number of windows increase the scheduling overhead. In addition, latency can be exploited over a window and not over the complete waveform. Optimal selection of window size is very difficult, so heuristic techniques are usually used. A window selection algorithm proposed by White [14] is given below.

The algorithm begins by selecting an initial estimate of the window size. The size of the next window is reduced if the number of points necessary to describe a waveform in the current window is more than a pre-defined limit. This limits the amount of storage necessary for waveforms. Similarly the size of the next window is reduced if the number of iterations required for convergence exceeds a pre-defined limit. Upon converge of the present window, the size of the next window is computed using the size of the present window and the maximum number of points required to describe a waveform in the previous window. A few optimizations to ensure that window boundaries lie on input breakpoints have been omitted from the algorithm for clarity.

Windowing Algorithm

```
start time = Beginning of window
stop time = End of window
endtime = End of user - defined simulation interval
usedpts = Max. number of points used the last window
prevwindow = Size of the window used in the previous iteration
if (Not entirely converged in this window)
 {
  if (usedpts ≥ max pts)
    {
     Shorten window if the waveforms overran storage buffers.
     stoptime = starttime + (prevwindow * max pts * 0.7) / usedpts;
    }
 else if ( numiters mod 5) == 0)
       {
        / * Half window size after every five WR iterations * /
        stoptime = prevwindow / 2 + starttime
       }
    else
       {
       / * Just do the same window * /
       stoptime = starttime + prevwindow;
       }
 }
else
 {
  / * New Window * /
   starttime = stoptime;
   stoptime = starttime + (prevwindow * max pts * 0.7) / usedpts;
 }
}
```

2.5.2 Circuit Partitioning

Waveform relaxation based simulators partition a given circuit into a number of sub-circuits. Circuit partitioning avoids solution of large sparse matrices. In addition, it is observed that the presence of even a few tightly coupled nodes in a circuit slows convergence. Therefore tightly coupled nodes can be isolated in sub-circuits and sub-circuits can be solved using a direct method.

Circuit partitioning techniques can be classified as static and dynamic. Static techniques perform *a priori* partitioning of a given circuit. Dynamic partitioning techniques re-partition the circuit during simulation. A dynamic partitioning implementation for simulation of bipolar circuits has been reported by Marong et al. [36] However dynamic partitioning techniques are rarely used in practice due to their complexity. Static partitioning schemes are described below.

Approaches which have been used for static partitioning of circuits can be classified as user partitioning, functional extraction, *dc* component (dcC) partitioning [37][38] and Norton equivalent conductance partitioning [10]. In the first scheme, the user specifies partitions. This scheme works well for many practical circuits. However users usually specify partitions from the design point of view which may not be ideal for the WR algorithm. Therefore some form of sanity check is necessary. The functional extraction method extracts functional blocks (gates, flip-flops) of a circuit to form sub-circuits. It is assumed that nodes of a functional block are tightly coupled; therefore they may be placed in a sub-circuit. This type of partitioning is difficult to perform, since the algorithm must recognize broad classes of functional blocks.

The dcC partitioning algorithm and Norton equivalent partitioning algorithm explicitly use coupling information for partitioning. Therefore these algorithms are likely to give better partitions for WR. The dcC algorithm uses circuit topology to determine coupling. According this algorithm, a set of elements is said to form a sub-circuit or dcC if there exists a direct path exclusively composed of two-terminal elements (resistors, voltage sources) and/or drain-source connections of MOS transistors between two nodes of that set. Further any two nodes of this set cannot be linked to any node of

the remaining circuit by such a path. This initial partitioning obtained by this algorithm is refined by breaking large sub-circuits and combining very small sub-circuits. A typical dcC implementation represents a circuit as a graph and the depth first search technique is used to obtain strongly connected components of the graph which represent sub-circuits. This technique is described in [37] and [38].

The Norton equivalent conductance partitioning algorithm uses estimates of the Norton equivalent conductance between nodes to partition a circuit. It is an extension of diagonally dominant loop criteria for partitioning linear systems which is explained below. Consider a system of equations of the form $f(x)=0$, where $x \in R^n$, $f: R^n \rightarrow R^n$ and x^k is generated by the kth iteration of the relaxation algorithm. Then the iteration factor γ is defined as the smallest positive integer such that

$$\left\| x^{k+1} - x^k \right\| \;\leq\; \gamma \left\| x^k - x^{k-1} \right\|, \tag{2.37}$$

for any $k > 0$, and any bounded initial guess x^0. The size of γ indicates the speed of convergence. If γ is much less than *1* then the relaxation converges rapidly. However, if γ greater than *1* then the relaxation may not converge. Consider a 2 dimensional system of equations given by:

$$\begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \end{bmatrix} \tag{2.38}$$

If the Gauss-Seidel method is used to solve the equation then the iteration factor is bounded by the spectral radius of the iteration matrix which is

$$\gamma^{gs} = \left| \frac{a_{21}a_{12}}{a_{11}a_{22}} \right|. \tag{2.39}$$

If both a_{12} and a_{21} are large, relative to a_{11} and a_{22}, then x_1 and x_2 are called tightly coupled variables. Similarly if both a_{12} and a_{21} are small then x_1 and x_2 are loosely-coupled variables. According to diagonally dominant loop partitioning criteria, two variables x_i and x_j are lumped if

$$\frac{a_{ij}a_{ji}}{a_{jj}a_{ii}} > \alpha. \tag{2.40}$$

where α is a constant.

The Norton equivalent conductance partitioning algorithm can be best explained with the help of an example electrical circuit [14, 8]. Consider the linear circuit shown in Figure 2.7. The circuit behavior can be described using a system of linear equations. The iteration factor for the conductance portion of the circuit is given by:

$$\gamma^{gs} = \frac{g_{12}}{(g_2 + g_{12})} \frac{g_{12}}{(g_1 + g_{12})}. \tag{2.41}$$

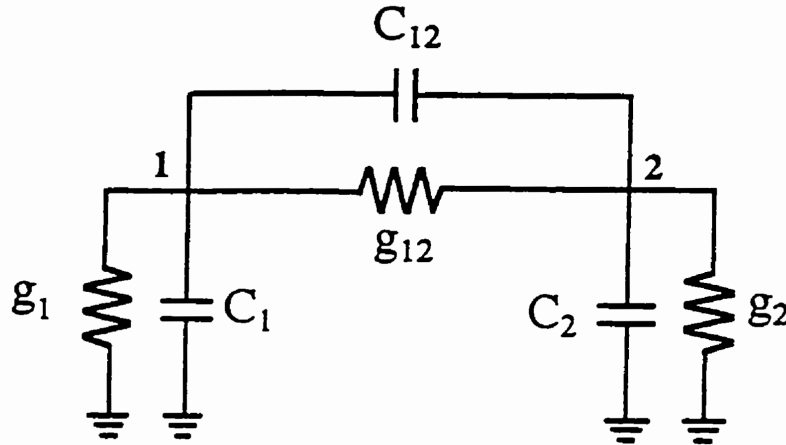


Figure 2.7: A linear Circuit Considered for Partitioning.

A similar expression can be written for the capacitance portion of the circuit. Two nodes are placed in a sub-circuit if the iteration factor exceeds a particular threshold. If nodes 1 and 2 are a part of a larger circuit then g_1 and g_2 are the corresponding Norton equivalent conductances. Heuristics used for computing Norton equivalent conductances of MOS circuits are given below.

Devices are replaced by their linear equivalent circuits. Nonlinear device conductances/capacitances and therefore their linear equivalents vary with inputs. The worst case values of conductances/capacitances are assumed for

partitioning. The computation of Norton equivalent conductance seen by a node involves tracing paths from that node to all other nodes in the circuit. Since the worst case conductance of a MOS transistor is zero, the trace is truncated whenever the gate of a MOS transistor is encountered. The resulting partitioning algorithm is given below [8].

```
g12 ← 0; g1 ← 0; g2 ← 0;
foreach (conductive elements between nodes 1 and 2 ) {
    g12 ← g12 + maximum element conductance over all v;
    Remove the element from the circuit;
}
g1 ← sum of the minimum Norton equivalent
    conductance of each element at node 1.
g2 ← sum of the minimum Norton equivalent
    conductance of each element at node 2.
if ( g12/(g2 + g12) · g12/(g1 + g12) > α) {
    Place the two terminal nodes in the same subcircuit;
}
```

A similar algorithm is written using capacitive elements and the union of the two results is used for partitioning. A disadvantage of this algorithm is that it may produce some very large sub-circuits. An additional partitioning pass is usually necessary to break large circuits and combine small circuits. It is also important to note that the partitioning criterion is very local.

2.6 Summary

This chapter has described the basic techniques for formulation of the circuit equations based on Kirchoff's laws. Direct and iterative techniques for solution of these equations have been described. Selection of an appropriate simulation time step is essential for reducing the simulation time while maintaining accuracy. Various techniques for selecting time steps have been described. Three iterative techniques namely: *linear relaxation, non-linear*

relaxation and *waveform relaxation* have been discussed. Speed-up techniques used in conjunction with the WR techniques were also outlined. The relaxation based techniques are ideally suited for implementation on parallel processors. The following chapter describes parallel relaxation methods.

3. PARALLEL RELAXATION METHODS

Relaxation techniques partition a circuit into a number of sub-circuits. Each sub-circuit is simulated independently using either a direct or an iterative technique, and an iterative technique is applied across sub-circuits. This makes relaxation techniques suitable for parallel implementation. Parallel relaxation methods are designed to exploit coarse grain parallelism across sub-circuits and fine grain parallelism in a single iteration of a sub-circuit. Appropriate selection of a parallel relaxation method depends on the nature of the circuit and the characteristics of the parallel architecture. This chapter describes parallel architectures and issues involved in implementing parallel applications. Parallel forms of waveform relaxation and iterated timing analysis are described. Partitioning and placement techniques used for implementing applications on distributed memory machines are also discussed.

3.1 Parallel Processing Techniques

The basic principle of any parallel processing system is to partition a given problem into a number of sub-problems and solve the sub-problems concurrently. As computing device characteristics approach their physical limits, it will become increasingly expensive to implement high performance uniprocessor systems. Therefore parallel processing techniques are increasingly used for the solution of CPU intensive problems such as image processing, three-dimensional fluid modeling and finite element analysis. More recently, due to commercial availability of multiprocessor systems, parallel processing systems have become attractive for the analysis and design of VLSI circuits. Important advantages of parallel processing systems include their low cost to throughput ratio and scalability. This section describes the issues involved in the design and application of MIMD computers. Shared memory and distributed memory architectures are

discussed [16][18]. The Kendell Square Research KSR-1 [39, 40, 41] is an example of a novel shared-memory multiprocessor. Important features of KSR-1 are described.

3.1.1 Shared-memory Systems

Shared memory architectures use a global, shared memory for inter-processor communication and coordination. The mechanism used for interconnecting processors and memory modules is an important architectural characteristic of shared-memory computers. Commonly used interconnection mechanisms are shared bus, crossbar switch and multi-stage inter-connection network [16].

A time-shared common bus is the simplest mechanism to interconnect processor and memory modules. It can be used for systems with moderate numbers of processors, ranging from four to *20*. Since only one processor accesses the bus at any given time, the bus bandwidth usually limits performance of the system.

The crossbar interconnection technology uses a crossbar switch of n^2 cross points to connect n processors with n memories (see Figure 3.1). The crossbar switch is a non-blocking network. It provides a dedicated path for communication between each processor-memory pair; therefore contention for communication links is avoided. Power, pinout, size and cost considerations have limited crossbar architectures to a small number of processors (from four to 16). The Alliant FX/8 is a commercial architecture that uses a crossbar scheme [16].

Multistage interconnection networks strike a compromise between price/performance alternatives offered by crossbars and buses. An $N \times N$ MIN connects N processors to N memories using multiple stages of switches. When N is a power of *2*, one alternative is to employ $\log_2 N$ stages of $N/2$ switches, using 2×2 switches. A significant feature of MINs is scalability. The BBN Butterfly multiprocessor used a MIN for connecting processors to memories. It could be configured with up to 256 processors [16].

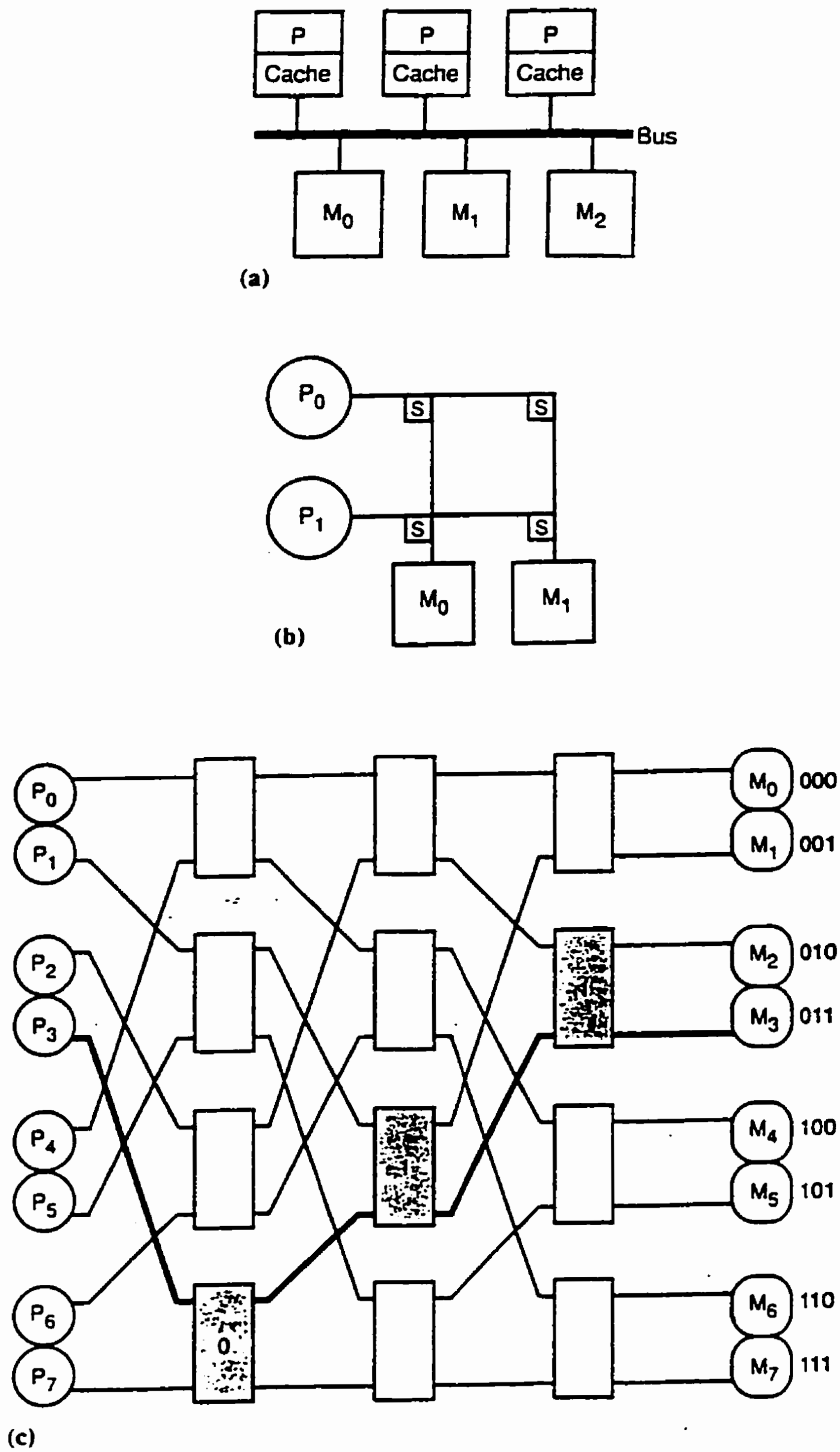


Figure 3.1: Shared memory interconnection networks [16]
a) Common bus b) Crossbar switch
c) Multistage interconnection

Each processor in a shared-memory architecture can address the global shared memory. This makes the shared memory programming model similar to the uniprocessor programming model. Parallel versions of commonly used programming languages such as parallel FORTRAN and parallel C allow conversion of existing sequential code to a form suitable for running on shared memory machines. In addition, lower inter-processor communication costs make this architecture suitable for exploiting fine grain parallelism. However important problems such as data access synchronization and cache coherency, must be solved. These problems are briefly described below. Coordinating processors with shared variables requires atomic synchronization mechanisms to prevent one processor from accessing a datum before another finishes updating it. The "test-and-set" is an example of a synchronization mechanism. It provides an atomic operation that subjects a key to a comparison test before allowing the key or associated data to be updated.

Typically each processor in a shared memory architecture also has a local memory used as a cache. Therefore multiple copies of the same shared data may exist in various processor's caches at a given time. Maintaining consistent versions of such data is the cache coherency problem. Cache coherency mechanisms provide new versions of cached data to each involved processor whenever a processor updates its copy. Small multiprocessor systems can use hardware "snooping" mechanisms to determine when shared data has been updated. Large systems usually depend on software mechanisms to ensure consistency.

3.1.2 Distributed Memory Systems

Distributed memory architectures connect multiple autonomous processing modules using a processor-to-processor interconnection network. Each processing module consists of a processor and its local memory. Processing modules (nodes) share data by explicitly passing messages through the interconnection network. Distributed memory computers are relatively less expensive and scale well. Various interconnection networks have been proposed to support scalability. In addition, certain classes of algorithms can be efficiently implemented using a specific interconnection topology. Several metrics are used to compare interconnection networks. Two

important characteristics of a network are node degree and network diameter. Node degree refers to the maximum number of communication links supported by a node and network diameter is the maximum number of communication links that must be traversed to transmit a message to any node along the shortest path. It is assumed that all nodes in the network are identical. Figure 3.2 shows commonly used topologies.

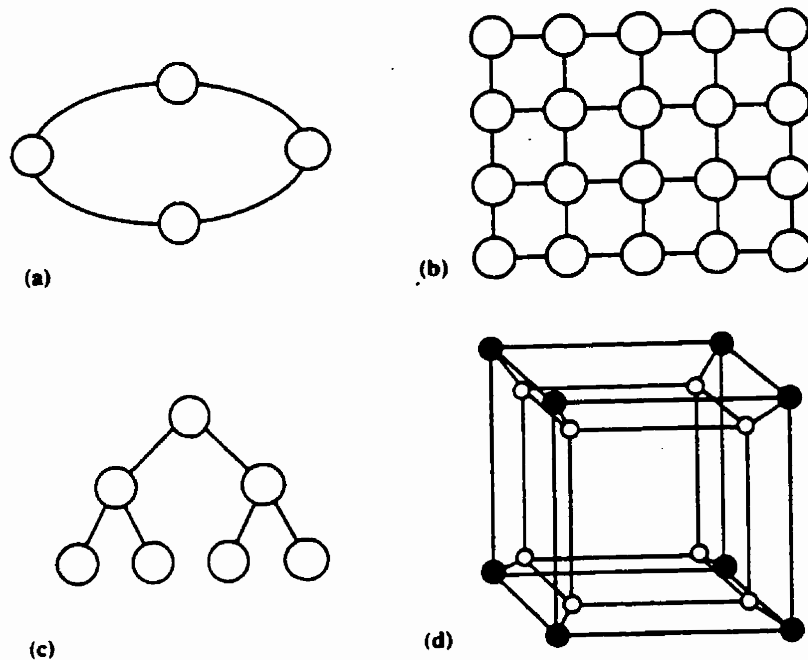


Figure 3.2: Multi-computer interconnection network topologies [16]
a) Ring b) Mesh c) Tree d) N-dimentional cube

Ring topologies are most appropriate for a small number of processors executing algorithms not dominated by data communications. A two dimensional mesh or lattice has n^2 nodes, each connected to its four immediate neighbors. Wrap-around connection can be provided at the edges to reduce the diameter of the network to $2 * (\lfloor n / 2 \rfloor)$. Communications can be augmented by providing additional diagonal links or by using buses to

connect nodes by rows and columns. Mesh topology architectures are used for matrix computations. Tree topology architectures have been constructed to support divide-and-conquer algorithms for searching, sorting and image processing applications. Strategies used to reduce the communication diameter of tree topology include adding additional links to connect all nodes at the small tree level. A hypercube topology uses $N = 2^n$ processors arranged in a n-dimensional cube [19]. Individual nodes are uniquely identified by n-bit numeric values ranging from *0* to *N-1* and assigned in a manner that ensures adjacent node's values differ by one bit. Hypercube architectures were developed to support performance requirements of 3D scientific applications. Examples of commercial hypercube implementations include the Ametek Series 2010, the Intel Personal Super-computer and the Ncube/10 [16].

The distributed memory programming model is substantially different from the shared-memory programming model. It is usually difficult to program distributed memory computers. Partitioning, allocation and load balancing is usually done by the application programmer. General forms of partitioning and allocation problems are NP complete [42]. These are discussed in Sections 3.8 and 3.9. Load balancing techniques are discussed in Chapter 5.

3.1.2.1 KSR Architecture

Kendell Square Research introduced *KSR-1*, a shared-memory multiprocessor that combines the advantages of conventional shared-memory and distributed-memory architectures [39, 40, 41]. The *KSR-1* can be configured with up to *1088* processors. The scalability of *KSR-1* can be attributed to a novel distributed memory scheme, ALLCACHE, which provides efficient mechanisms for exploitation of locality. Work is not bound to a particular memory, but moves dynamically to available processors. Hardware support is provided for reducing access time. The *KSR-1* architecture is briefly described below.

The *KSR-1* system is designed using a hierarchy of slotted rings. The lowest level in the hierarchy, *level 0* ring, operates at 1 GB/sec (128 million accesses per second) and connects 32 processor cells (see Figure 3.3). Each processor

cell consists of a 64-bit superscaler processor, 32-Mbytes local cache and a local cache directory. The processor cell consists of a search engine which provides hardware support for migrating data to and from other nodes and provides memory coherence throughout the system using distributed directories and for ring control. The *level 1* ring is used to connect 34 *level 0* rings. A standard *KSR-1 level 0* ring consists of 34 slots: 32 for the processors and two for the directory cell connected to the *level 1* ring. Each slot can be loaded with a *subpage* consisting of 16-byte header and 128 byte of data.

The ALLCACHE design of *KSR-1* eliminates memory hierarchy and the corresponding physical memory addressing overhead. It represents a confluence of cache and shared virtual memory concepts. The KSR machine provides a strictly sequential consistency programming model. In this model, every processor returns the latest value of a written value. Therefore results of an execution on multiple processors appear as some interleaving of operations of individual nodes when executed on a multi-threaded machine. ALLCACHE mechanisms also provide hardware support for memory management though migration and replication of data.

The *KSR-1* provides three levels of cache access: an intra-node, an inter-node with the responder processor/cache cell connected to the same *level 0* ring, and an inter-node with the responder processor/cache cell connected to a different *level 0* ring. The inter-node communication for remote cache access is done through a searching process. When the requester and responder are connected to the same *level 0*, a local cache directory provides the local cache-access reference. When the requester and responder are connected to a different *level 0*, the request is communicated using *level 1*. The *level 1* consists entirely of ring routing cells. Each ring routing cell contains the directory of the *level 0* ring connected to it. These directories are used for routing the request to the appropriate *level 0* ring and subsequently to the appropriate cell.

The KSR system uses a Mach-based operating system. The multi-programmed operating system allows users to run multi-process multi-threaded applications. The *KSR-1* also provides a scalable commercial programming environment for transaction processing that accesses relational databases in parallel .

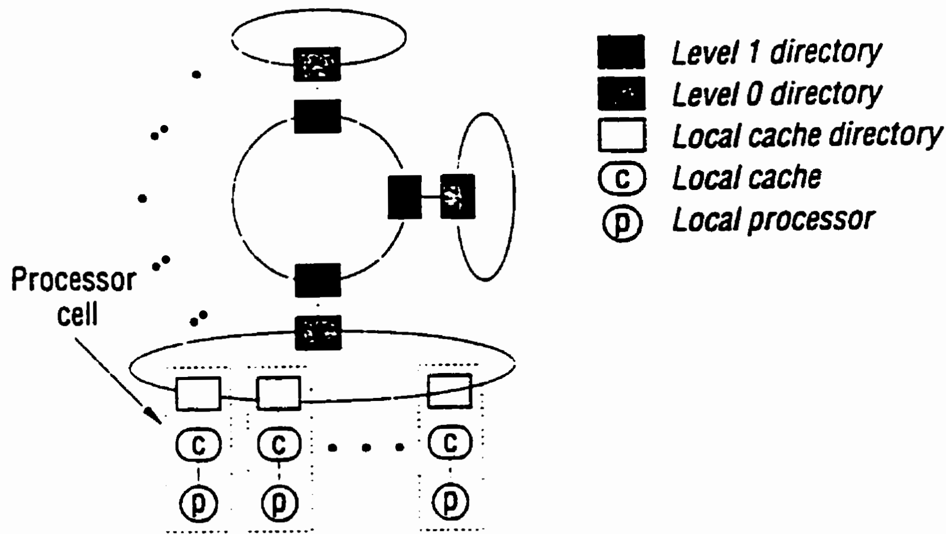


Figure 3.3: KSR-1 architecture with a slotted ring for communication [39].

3.2 Efficiency and Speed-up in Parallel Systems

To exploit the power of a multiprocessor computer, as much of the computation as possible should be performed concurrently. In order to understand how to partition the circuit simulation problem to get maximum concurrency, it is necessary to estimate how much faster an N processing element (N-PE) machine is, compared to a single processing element machine.

The speed-up of an N-PE machine is defined as

$$S = \frac{t_1}{t_N}, \tag{3.1}$$

where t_1 is the time required for solving a problem using one PE, and t_N is the time required to solve the problem using N concurrent PEs. An ideal case is linear speed-up, i.e., an N fold reduction in computation time resulting from an N fold increase in the number of processors $t_1 = N * t_N$. However, it is

usually not possible to run the entire program in parallel. Therefore, with α the portion of the program requiring sequential execution,

$$t_N = t_1(\alpha + \frac{1-\alpha}{N}) \tag{3.2}$$

Now as N tends to infinity,

$$\lim_{N \to \infty} S = \frac{1}{\alpha} \tag{3.3}$$

This is known as Amdahl's law, and can be stated in the following form: The reciprocal of the fraction of the computation that must be done sequentially limits the number of processors that can usefully be put to work on a given problem [43].

From Amdahl's law it is evident that to use many concurrent processors, the sequential fraction must be small. The situation discussed above depicts an ideal case in which the parallelizable part can be equally split between N PEs. However, in practice, load imbalance may result in partially idle processors. Communication overhead also contributes to extra work performed by each processor. Collecting the idle time and the overhead time into one parameter, σ, and applying it to t_N,.

$$t_N = t_1(\alpha + \frac{1-\alpha+\sigma}{N}), \tag{3.4}$$

which yields, if inserted in Equation 3.1

$$S = \frac{N}{1+\sigma+\alpha(N-1)}. \tag{3.5}$$

In the case when σ is independent of N, and α is very small compared to N, it is possible to use each computing element with an efficiency of

$$\frac{S}{N} = \frac{1}{1+\sigma}. \tag{3.6}$$

that is, each processing element will spend a fraction S/N of the elapsed time in solving the actual problem.

The assumption that σ is independent of N is simplistic. However, it can be observed that an algorithm that causes some extra work, i.e. has large σ, can still be useful if it has low α. A fairly low efficiency can be tolerated as long as t_N decreases with an increase in the value of N.

3.3 Parallel Direct Methods

Most relaxation based simulators use direct methods for simulation of sub-circuits. Therefore it is important to study the parallelism in the direct method. An algorithm for performing transient analysis using the direct method is presented in Section 2.2. The majority of time spent in performing transient analysis is lumped in two categories: time required to assemble a system of linear equations for each iteration of Newton's method at each time point, the FORM phase, and the time required to solve the system of sparse linear equations, the SOLVE phase. The FORM phase requires linearization of the nonlinear element characteristics and the addition of various conductances, currents and charge values into the Jacobian and the right-hand-side (RHS) vector. The SOLVE phase typically involves LU decomposition to solve the system of equations. Several researchers have observed that for small circuits, with number of nodes less than *20*, the majority of the time is spent in performing the FORM. However, when the size of the circuit grows, an increasing percentage of the time is spent in the SOLVE phase. Therefore it is necessary to study parallelism in both phases. The actual parallalism that can be effectively exploited depends on the nature of the circuit, the complexity of device models, and the characteristics of the target parallel computer. Strategies for exploiting parallelism in the FORM and the SOLVE phases using shared-memory and distributed-memory architectures are described below.

The contributions of each device to the Jacobian and the RHS can be computed independently. Therefore parallelism is inherent in the FORM phase. However, this is a fine grained parallelism and it is necessary to ensure that overheads do not offset gains due to parallel implementation. As

the contributions are computed they have to be added to the Jacobian and the RHS vector. This accumulation is a serial process. Its implementation on shared-memory machines requires synchronization to ensure that only one processor is updating a particular element of the matrix at a time. Lock-based or barrier-based methods can be used for synchronization. Lock-based methods use a lock on some region of data, for example, per element, per row or one for the entire matrix. Processors accumulate contributions of devices to the Jacobian and RHS in local storage assigned to each instance of a model. A lock is seized before making an update to the region associated with it. An increase in the number of locks reduces the likelihood of processor contention at the cost of increasing the locking overhead. A lock-per-row scheme is usually considered a reasonable compromise. Barrier-based methods transform distributed lock synchronization points into one or more barriers which separate the sequence of evaluation and accumulation operations. A matrix template is allocated for each device in global memory. Therefore, contributions of all devices can be computed and stored in parallel. A single synchronization point is used at the end of calculations. After the synchronization point, the Jacobian and RHS can be updated sequentially or in parallel. The main drawback of this approach is the increased storage requirement.

Several different parallel device model evaluation schemes are possible using distributed memory computers. An appropriate selection scheme depends on the size of the circuit, the size of the primary and secondary storage associated with a node, and computation-to-communication ratio of the device model evaluation task. A basic process farming scheme and its variants are described below. The process farming scheme divides the pool of processors into a single farmer and multiple workers. The farmer stores Jacobian, RHS vectors, and a queue of devices. The farmer sends the device information (e.g. instance specific model information, node voltages, node charges) to a free worker which computes contributions of the device to the Jacobian and the RHS and sends it back to the farmer. The farmer updates the Jacobian and the RHS. A linear array and m-ary tree topologies are commonly used for implementing a process farming scheme. The m-ary tree topology is useful for reducing the average distance between a farmer and a worker. The process farming approach works well on architectures that can overlap

computation with communication. Buffers are provided on each node to improve utilization of worker processors. It is also possible to organize the process farm as a single master, multiple farmers and each farmer associated with a set of workers. This arrangement also distributes templates of the Jacobian and the RHS to farmers. Farmers do partial accumulation of the Jacobian and the RHS.

Parallelizing the SOLVE phase involves parallel solution of a system of asymmetric sparse linear equations. Parallelizing the SOLVE phase is complex compared to parallelizing the FORM phase due to dependencies involved in Gaussian elimination. This problem has been widely studied in the literature and is an active area of research [25][44][45]. A brief discussion of issues involved in parallel sparse system solution is given below.

The linear equation solution is usually performed by using LU factorization followed by forward elimination and backward substitution. A typical form of LU decomposition consists of the following steps. The row/column associated with the diagonal element a_{11} is divided by the pivot element (a_{11}) and then each element a_{ij} in the lower right corner of the matrix is updated by subtracting the product $a_{1j} * a_{i1}$. This is followed by the division and updating (factorization) of $a_{22}, a_{33}, \cdots, a_{nn}$ until the entire matrix is factorized.

Parallelism involved in the LU decomposition process can be classified as *fine grained*, *medium grained,* and *coarse grained*. In *fine grained* parallelism division of all elements in a pivot row are performed in parallel. This is followed by a parallel update of all elements in the particular row/columns involved in the factorization step. In *medium grained* parallelism, the operations associated with two or more rows are performed in parallel. Fine and medium grained parallelism are considered appropriate for implementation on vector processors. The efficiency of these approaches depends highly on the architecture and the overheads of the implementation. *Coarse grained* parallelism is associated with independent pivots. This form of parallelism can be exploited using shared-memory and distributed-memory machines. It is described below.

A pivot a_{jj} is dependent on pivot a_{ii} if a_{jj} must be factored after a_{ii} to guarantee a correct solution. The dependence can be direct or indirect, that is, a_{jj} may depend on some other pivot which in turn is dependent on a_{ii}. Computations associated with independent pivots can be performed in parallel, however, it is important to note that shared-memory implementations require appropriate synchronization mechanisms to ensure correctness of solution. Most parallel implementations which exploit coarse grain parallelism, reorder the matrix to increase the number of independent pivots. This pivot reordering can conflict with the conventional sparse matrix reordering (e.g. Markowitz ordering) done to reduce the number of operations. Therefore appropriate selection of a reordering scheme that balances the increase in parallelism against minimizing fill-ins is very important. Block structured approaches such as nested dissection [46] and sub-structuring [47] can be used for this purpose. Examples of parallel circuit simulators that exploit this form of parallelism are SUPPLE [48] and PECSI [49].

3.4 Common Parallelism In Relaxation Methods

An objective of this section is to analyze parallelism common to relaxation methods. Issues involved in implementing relaxation based simulators on shared memory machines are reviewed. The discussion in this section is based on the work of Saleh et al. [27].

A closer look at ITA (nonlinear relaxation) and Waveform relaxation reveals that two forms of parallelism exist in both methods: coarse gain parallelism across sub-circuits, and fine grain parallelism within a single Newton iteration of a particular sub-circuit, i.e., some sub-circuits can be evaluated in parallel and the sub-circuit evaluation process itself can be broken down into small sub-processes. Some of these sub-processes (not necessarily all) can be executed concurrently.

The computations involved in the solution of a single Newton iteration are similar to those in the standard method. Therefore, the problem of parallelizing a single Newton iteration is equivalent to the problem of parallelizing the direct method. Considering the small size of each sub-

circuit, one that may contain even a single circuit node, parallel model evaluation is the only form of parallelism available at the finest level of granularity. The largest-grain parallelism exists at the sub-circuit level, and the amount of parallelism available depends on the particular relaxation scheme used and any additional synchronization points that are introduced or removed for architectural and programming reasons.

The first task performed by a parallel circuit simulator is to partition the circuit and generate a sub-circuit graph. The sub-circuit graph contains a vertex for each sub-circuit and the edges represent the dependency relationships between sub-circuits. There is a directed edge from vertex i to vertex j, if sub-circuit j contains an equation that depends on the value of the node voltage in sub-circuit i. The edges represent the flow of signals between sub-circuits. Thus the sub-circuit graph originates from primary inputs and terminates on the final outputs.

The sequence of scheduling sub-circuits (partial ordering information) can be extracted from the sub-circuit graph. If the sub-circuit graph is acyclic, then the circuit can be solved by exactly one relaxation iteration, provided that the solution of a sub-circuit is not started until the solution of its fan-in sub-circuits is completed. MOS circuits without feedback, that use simplified transistor models, where the gate-to-drain and gate-to-source capacitances have been omitted, can result in acyclic sub-circuit graphs.

Circuits with feedback and complex device models result in sub-circuit graphs that contain cycles, i.e. a directed edge from sub-circuit i to j as well as an edge from sub-circuit j to i. This directed cyclic graph can be converted to a directed acyclic graph (DAG), by retaining only one edge which corresponds to the dominant direction of signal flow (usually the one in the feed forward direction). The DAG specifies the data dependence to be enforced in one relaxation iteration. It is also the sub-circuit task dependence graph which determines the exploitable parallelism at the sub-circuit level.

Consider the Gauss-Seidel iteration as an example. The first step in determining the partial ordering is to break any global feedback loops by deleting one or more of the edges in the loop and converting them into cross

iteration edges. In order to understand the process, consider three circuits A, B, and C. Outputs of A are connected as inputs to circuit B and outputs of circuit B are connected as inputs to circuit C. There exists a global feedback from circuit C to A. For generating a DAG, the global feedback from C to A is broken and the output from the *k*th iteration of C is treated as an input to the *k+1*th iteration of A. The *n* sub-circuits are thus partially ordered into *m* ranks based on the distance in terms of the directed edges from the input source.

As mentioned earlier, the partial ordering defines the data dependence of a single Gauss-Seidel iteration. Specifically, a circuit can execute its *k*th iteration when all the sub-circuits of lower rank fanning into it by a directed edge have completed their *k*th iteration and all the sub-circuits of higher rank fanning out from it have completed their *k-1*th iteration. The Gauss-Jacobi method is generated by placing all the sub-circuits in a single rank and ignoring directionality of the circuit within a single iteration. Figure 3.4 shows the examples of Gauss-Seidel and Gauss-Jacobi partial ordering for a single iteration. The Gauss-Seidel (GS) graph has three ranks and the Gauss-Jacobi (GJ) graph has only one rank.

Figure 3.4 shows sub-circuit level parallelism. Any circuit in the same rank can be processed simultaneously, leading to an implementation known as the multiple barrier approach. The implementation essentially consists of a sequence of *m* DOALL loops corresponding to the *m* ranks of the task graph. Artificial synchronization points (barriers) can be introduced between two ranks in order to exchange essential global information and take convergence decisions. The approach is attractive for circuits that have adequate activity in each rank and can be easily implemented on machines that have hardware support for loop-based parallelism including the DOALL construct. However, latency of sub-circuits and dramatically different sub-circuit sizes can make this approach less effective. In such cases, the use of synchronization points between ranks can suppress parallelism that is inherent in the relaxation scheme.

One way of overcoming this problem is to statically alter the partial ordering, i.e., shift sub-circuits from one rank to another at compile time. This approach has its limitations. The limiting case is the Gauss-Jacobi technique

in which all the sub-circuits are in the same rank. The obvious disadvantage of this approach is a possible reduction in convergence rate. Therefore, factors such as number of available processors, coupling between sub-circuits, the effect of the change on convergence rate, latency, and the sub-circuit task sizes must be considered before altering the partial ordering.

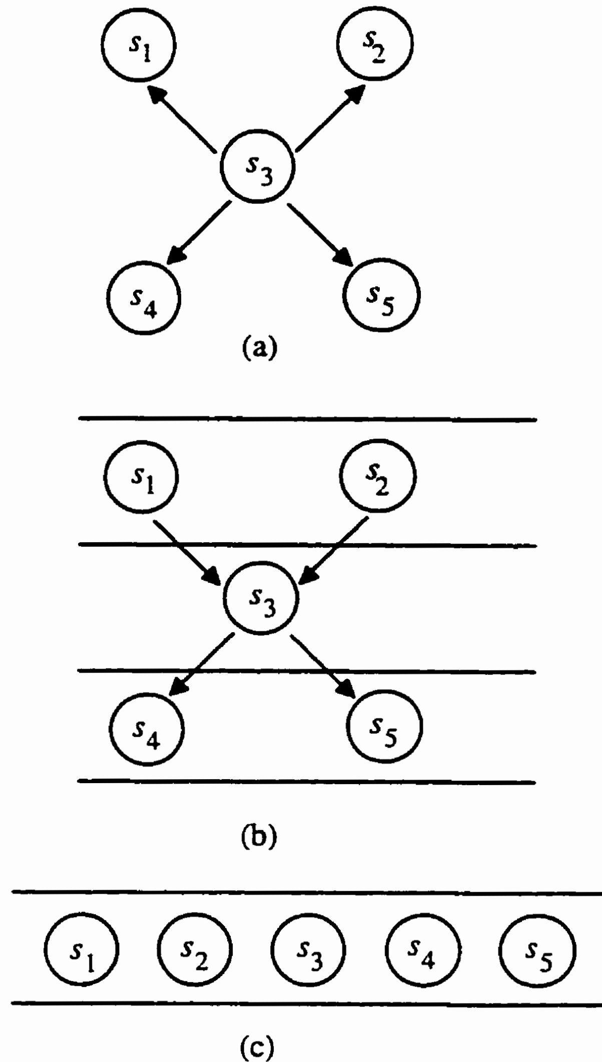


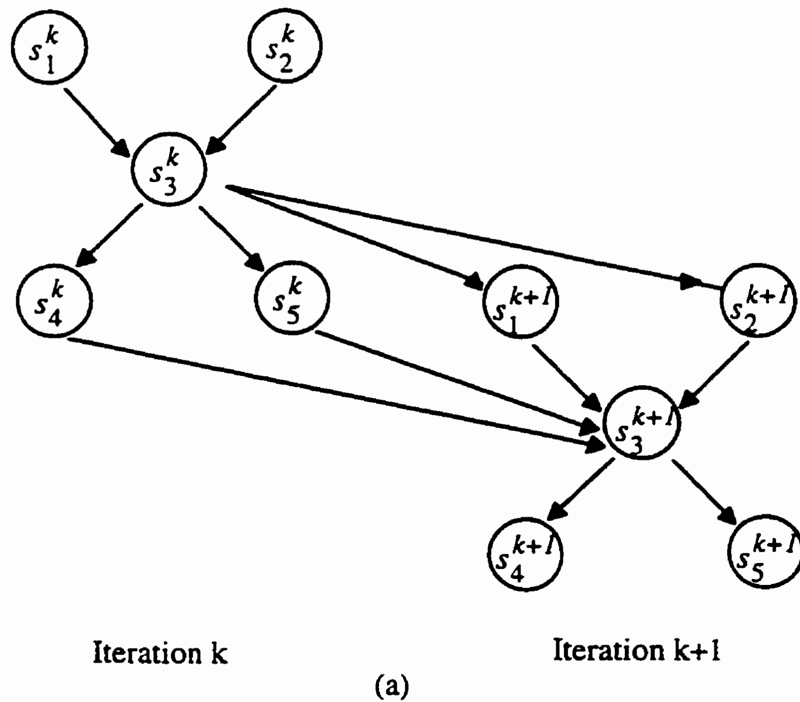
Figure 3.4: Examples of Gauss-Seidel and Gauss-Jacobi partial ordering for one iteration (a) Original sub-circuit task graph (b) Gauss-Seidel ordering m=3 (c) Gauss-Jacobi ordering m= 1 [27].

Another way to improve the multiple barrier approach is to remove artificial synchronization points between ranks and introduce only a single barrier synchronization point at the end of each iteration. This barrier ensures that all the sub-circuits have completed their kth iteration before starting the $k+1$th iteration. This modification involves processing the sub-circuits using a dataflow driven scheduling mechanism. When a processor gets a sub-circuit task, it waits for the dependencies of the assigned sub-circuit task to be satisfied before proceeding with the task. Although this mechanism is conceptually straightforward, it implies a more complicated implementation of the control mechanism. Typically an efficient queue based scheduling mechanism is essential. It is important to note that Gauss-Jacobi single-barrier and multiple-barrier versions are identical because all the sub-circuits belong to a single rank.

The multiple barrier and single barrier approaches are summarized below. A multiple-barrier Gauss-Seidel approach divides the sub-circuits in several ranks based on their distance from the primary inputs. When all the sub-circuits in the mth rank (always starting with the first rank and first iteration) complete the kth iteration, sub-circuits in the $m+1$th rank are scheduled for execution of the kth iteration. Upon completion of the kth iteration by all the sub-circuits, the sub-circuits in the first rank are scheduled for performing the $k+1$th iteration and the process is continued until convergence. In the single-barrier Gauss-Seidel approach, execution of the kth iteration continues in a data driven manner. When all the sub-circuits complete the execution of the kth iteration, the $k+1$th iteration is scheduled for execution. In both the techniques described above, at any given time, only one iteration is available for execution.

The amount of parallelism in the above mentioned approaches can be further increased by removing the restriction that only one iteration can be scheduled for execution at any given time, i.e., by eliminating the artificial barrier between the iterations. The $k+1$th iteration of tasks can be scheduled for execution as soon as the appropriate tasks in the kth iteration are completed. This can be viewed as "unrolling" the data dependence graph as illustrated in Figure 3.5. The convergence decisions and update of global information that were performed at the synchronization points are now

distributed. This results in a fairly complicated thread of control and requires priority queue based scheduling mechanism to ensure that tasks from earlier iterations take precedence. The return on the increased complexity is circuit and architecture dependent. This scheme is feasible only if unrolling of a limited number of iterations is allowed.



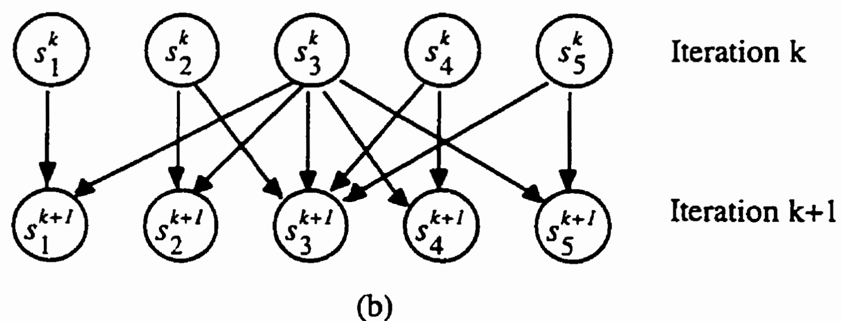


Figure 3.4: Unrolled (a) Gauss-Seidel (b) Gauss-Jacobi iterations [27]

Another important issue common to most parallel implementations is the determination of the appropriate size of individual sub-circuits. Large variation in sub-circuit sizes may create load imbalance among processors. One way to deal with this problem is to perform partitioning in two passes. The second pass can combine small sub-circuits and/or break large sub-circuits. The effect of combining two smaller sub-circuits into one large sub-circuit is two-fold. First the amount of parallelism is reduced in favor of making the task sizes uniform. Second, the efficiency of circuit latency exploitation is reduced since the probability of having at least one active node in a large sub-circuit is higher. However, improvement in the execution speed may offset the two negative consequences of combining sub-circuits, making it a worthwhile alternative. The second option of breaking up larger sub-circuits into smaller ones offers a trade-off of convergence speed for parallelism. The specific application of the two approaches depends on the number of processors available in the system.

3.5 Parallel Waveform Relaxation

As discussed in Section 2.5, the waveform relaxation technique applies the relaxation techniques at the differential equation level. The WR algorithm computes node waveforms over the complete time interval of interest (window). Node waveforms are exchanged and the process is iterated until convergence. The WR algorithm can be implemented in conjunction with the circuit partitioning and parallel processing techniques described in Section 2.5.1. In this case, the largest-grain task consists of solving a sub-circuit over an entire time window for one relaxation iteration. This is known as the full window technique and is described with the help of an example of a Gauss-Seidel single-barrier implementation.

A sub-circuit evaluation task in the current window $[T_{(start)}, T_{(stop)}]$ begins executing only after all of the waveforms from the fan-in tasks are computed over the same window interval. When a sub-circuit has computed its internal waveforms, it checks to see if any fan-out circuits are ready for execution. If so, these fan-out circuits are scheduled for processing. In this scheme, sub-circuit evaluation is treated as an indivisible entity. Since the sub-circuit evaluation task typically involves large amounts of computation,

the granularity of each task is very high and therefore task scheduling is relatively inexpensive. However, this approach does not fully exploit parallelism. It becomes inefficient if the number of processors is large or the sub-circuit task graph is long and narrow and does not provide enough parallelism to keep all the processors busy.

The time-segment and time-point [27] pipelining approaches provide more parallelism by simultaneously processing sub-circuits in different ranks. The increased parallelism is at the expense of increased overhead and a more complicated thread of control. The basic principle behind these approaches is to allow sub-circuits to begin computing their waveforms as soon as adequate information is available, instead of waiting for the fan-in circuits to complete computations for the entire window. For example, it is possible for a sub-circuit to compute its internal waveforms up to time t, $[t < t_{stop}]$, if all its fan-in sub-circuits have computed the required waveforms until time t. This form of parallelism can be exploited to various degrees by adjusting the number of time-points that a sub-circuit computes. If the sub-circuits are allowed to begin computing whenever their fan-ins have computed one new solution point, it is referred to as time-point pipelining. If a fixed number of time-points are computed before propagation occurs then the technique is referred to as time-segment pipelining.

Several different implementations of WR algorithms are possible, each having its own characteristic features and trade-offs. For example, GJ and GS algorithms trade parallelism for convergence rate. It has been reported that GS generates a set of computations that generally converge to the solution in fewer iterations than GJ [34]. However, the GJ method generates a higher degree of parallelism, as all the sub-circuit tasks of a given iteration can be executed concurrently. In the case of full window and time-point pipelining schemes, trade-offs exist between parallelism and overhead. Full window techniques lie on one end of the scale, with the least overhead and parallelism, whereas the time-point pipelining scheme lies at the other end of the scale, with a high degree of parallelism obtained at the expense of a large overhead. The time-segment pipelining scheme encompasses all the intermediate granularities between the two extremes. The appropriate choice of algorithm that would provide maximum possible speed-up depends on the

features of the target architecture and the nature of the circuit being simulated. In particular, the number of available processors, the costs of memory references, message passing costs, locks and other sources of overhead in the implementation, and the topology of the task graph all affect maximum possible speed-up. For example, for a circuit having a number of processors equal to the number of sub-circuits in each rank, with tasks of approximately uniform size, a full window technique using Gauss-Seidel approach would be appropriate. For a massively parallel machine with a relatively low communication overhead, an algorithm that provides the maximum amount of parallelism such as the Gauss-Jacobi approach with time-point pipelining would be most suitable.

Ideally, a pre-processing step in a parallel relaxation program would automatically determine the appropriate form of relaxation algorithm and degree of pipelining, based on the characteristics of the architecture and the circuit being simulated. This is usually quite difficult in practice. Although a rough estimate of speed-up can be obtained from the task graph, *a priori* determination of task sizes is usually difficult. This is due to the unpredictable nature of latency characteristics and the relaxation convergence speed. Therefore, post-simulation estimates, based on the information collected during uniprocessor simulations of a circuit, are used to predict the ideal speed-up of a particular algorithm on a varying number of processors.

Saleh et al. [27] have reported development of a program, PARASITE (PARAllel SImulation Timing Estimator), to estimate parallel execution times for any combination of pipelining and relaxation on a given number of processors. After performing a circuit simulation for a particular form of relaxation on a uniprocessor system, a weighted task graph is constructed for the form of pipelining specified. PARASITE takes a task graph, the CPU times associated with the tasks, and the number of processors used, as inputs. It mimics the operation of the specified parallel waveform relaxation algorithm, but instead of performing the computations to solve the circuit equations, it simply keeps track of the time that would be required to execute the tasks on a specified number of processors. The speed-up computed in this manner is an approximate upper bound because PARASITE neglects the

overheads. It is also possible to vary the degree of pipelining. The estimates obtained from PARASITE can be compared with the actual execution time of a parallel circuit simulator to determine the efficiency of the implementation.

3.6 Parallel Iterated Timing Analysis

As discussed in Section 2.4, Iterated Timing Analysis is a circuit simulation approach based on nonlinear relaxation. It uses a selective trace technique for latency exploitation. The simplified static scheduling models described in the previous subsection have a number of limitations when latency is exploited. It is difficult to predict which sub-circuits will become latent at the next simulation time point. Even if the sub-circuit task graph is updated at every simulation time point, it may be difficult to anticipate the sub-circuits that will become latent or be activated, due to fan-out scheduling, during the iteration process. Updating the task graph at every iteration at a time point involves considerable overhead and requires larger circuits than are often appropriate in ITA.

The above mentioned problems can be considerably reduced by appropriately selecting the subset of sub-circuits that can be processed on a given iteration. It is also important to decide when the selected sub-circuits can be scheduled for execution and the priority rule applied while scheduling. This can be best explained with the help of the following example. Consider implementation of the single barrier Gauss-Seidel approach. It can be modified so that a sub-circuit does not have to wait for all its external connections to satisfy the Gauss-Seidel scheduling conditions. An extreme form is used in the event-driven selective trace technique [3]. This technique treats the sub-circuit graph as a signal flow graph. Each vertex has a fan-in and fan-out table. Whenever the value of an input node or any internal node changes, an event is generated. As a result, all its fan-out sub-circuits are scheduled for processing. Subsequent events generated after processing the fan-out sub-circuits cause their fan-out sub-circuits to be scheduled for processing. The only circuits that are processed are those which are directly affected by the change. The order of processing sub-circuits is a function of the order of signal flow in the network and it therefore constitutes dynamic ordering. Whenever the fan-out sub-circuits are scheduled, they are placed on a queue

with priority based on the ranking due to dominant edges of the Gauss-Seidel task graph. The circuit may be scheduled more often than a sequential case.

3.7 Static Partitioning Techniques

The problem of solving a single problem using multiple processing units consists of two parts. The first part is to partition the problem into a number of task sets in such a way that maximum possible parallelism can be exploited. Partitioned task sets usually communicate with one another. The second part is to assign these task sets to interconnected processors. The objective of partitioning and allocation phases is to minimize run time by minimizing inter-processor communication and load imbalance overheads. The objectives of minimizing communication overhead and maximizing parallelism are usually conflicting. Similarly the requirement of minimizing communication overhead and load imbalance conflict with one another.

Static partitioning refers to *a priori* assignment of tasks to processors. Dynamic scheduling refers to runtime allocation of tasks to processors. Static partitioning schemes are commonly used for programming distributed memory machines, while most shared memory implementations use dynamic scheduling schemes. Shared memory machines typically maintain job queues common to all processors in the shared memory and free processors obtain jobs from the job queue during runtime. Dynamic scheduling is economical due to shared data structures and relatively low inter-processors communication costs. A similar arrangement is difficult in distributed memory machines due to a lack of shared address space and relatively high inter-processor communication costs, therefore *a priori* assignment of tasks to processors is commonly done. Several researchers have combined the static partitioning and allocation problems [50, 51, 52]. This section describes issue related to partitioning and to the partitioning-allocation combination. The following section describes allocation strategies.

Two approaches are commonly used to solve the partitioning and allocation problems. The first approach involves the use of domain-specific heuristic. This approach works well in situations where the parallel application is well structured and the geometry of the problem can be used for partitioning.

Examples of domain specific heuristic are box-wise decomposition, strip-wise decomposition and scattered decomposition. Domain-specific heuristics are commonly used for parallel solution of partial differential equations and iterative solution of sparse linear systems. The second, more general approach, is based on a mathematical cost function. The mapping obtained using this approach attempts to minimize the cost function. The domain specific heuristic schemes are usually computationally efficient. The cost function based schemes are often computationally time consuming, but more generally applicable and potentially capable of obtaining better mappings. Task graphs of relaxation based circuit simulation problems are usually unstructured. Therefore cost function based schemes are appropriate for this application. A formal statement of the partitioning/allocation problem [53] and a brief description of cost function based schemes are given below.

The parallel program is characterized by a task graph G(T,E), whose vertices, $T=\{t_1, t_2, \cdots, t_n\}$, represent the tasks of the program, and edges E, correspond to the data communication dependencies between those tasks. The weight of a task t_i, denoted w_i, represents the computational load of the task. The weight of an edge e_{ij} between t_i and t_j denoted as c_{ij}, represents the relative amount of communication between the two tasks.

The parallel computer is represented using a processor graph $G(P, E_p)$. The vertices, $P=\{p_0, p_1, \cdots, p_k\}$, represent the processors and the edges represent the communication. The system consists of homogeneous processors and communication links. The cost of communication is assumed to be proportional to the size of the message and the distance between sender and receiver. The distance d_{qr} between processors p_q and p_r is defined as the length of the shortest path between p_q and p_r.

The function $M: T \to P$ maps a task t_i to processors. The task set (TS_q) of a processor p_q is the set of tasks mapped onto it:

$$TS_q = \{t_s | M(t_s) = p_q\} \qquad q = 0, 1, \ldots, k \qquad (3.7)$$

The work load (WL_q) of processor p_q is the total computational weight of all tasks mapped on to it:

$$WL_q = \sum \{w_j | M(t_j) = p_q\} \qquad q = 0,1,\ldots,k \qquad (3.8)$$

The communication load (CL_q) of processor p_q is the total weighted cost of edges in its communication set, where each edge is weighted by the physical path length to be traversed under the mapping M:

$$CL_q = \sum_{r=0}^{k} \sum \{c_{ij} * d_{qr} | M(t_i) = p_q \quad and \quad M(t_j) = p_r\} \quad q = 0,1,\ldots,k \qquad (3.9)$$

The estimated parallel program time is used as the cost function for optimization. It is the completion time of the processor that completes tasks assigned to it last. If $T_e(p)$, $T_c(p)$, and $T_i(p)$ are the computational execution time, the time spent for communication and the idle time for processor p, then the total program completion time is given by:

$$T(p) = T_e(p) + T_c(p) + T_i(p) \qquad (3.10)$$

If the task execution times are known then $T_e(p)$ can be accurately modeled using WL_p. Although accurate modeling of $T_c(p)$ is difficult, it can be estimated with reasonable degree of accuracy. The idle time $T_i(p)$ is the most difficult to model. It depends on synchronization delays during program execution. If $T_c(p)$ is modeled using CL_p then the parallel program execution time can be expressed as:

$$T_M = MAX_p[T_e(p) + T_c(p)] = MAX_p[k_e * WL_p + k_c * CL_p] \qquad (3.11)$$

where weights k_e and k_c represent different relative time requirements for a unit of computation and a unit of communication. The idle time can be ignored because the maximum among all processors of the sum of communication time and computation time is calculated. The objective of any mapping function is to minimize T_M. This is know as a "*minmax*" approach. The minmax approach lumps all communication costs incurred in multi-hop communication and associates them with the sender of the message. The sum cost approach described below avoids this unrealistic assumption. According to this approach an *ideal* mapping distributes computational load uniformly among all processors and no communication cost is incurred. The summed cost approach expresses the cost function as:

$$cost = Penalty\ for\ computation\ imbalance + Penalty\ for\ communication$$

The penalty of computation imbalance is calculated as the sum among all processors of (the absolute values of) the deviation of the actually assigned load and the ideal average load. The total communication load in the system represents the penalty for communication. Thus the cost function is given by

$$cost = k_e * \sum_{i=0}^{k} |WL_i - AVGload| + k_c * \sum_{i=0}^{k} CL_i \qquad (3.12)$$

Partitioning is a difficult problem because an optimal solution must be selected out of the k^n possible assignments that arise when n tasks are assigned to k processors. This problem has received generous attention in the literature. Algorithms which yield true optimal solutions in the absence of resource constraints are well known to be NP-complete [42]. The approaches used for partitioning can be divided into three categories: graph theoretic approach [54, 55, 56, 57], mathematical programming approach [58], and heuristic approach [50, 51, 52, 53]. A brief description of these approaches is given below. A comparison of these approaches is also presented at the end of the discussion.

Stone [54, 55] and Bokhari [56, 57] have conducted several studies of the task assignment problem for non-precedence constrained task systems with an objective of minimizing total execution time and communication costs. Their work is mainly based on the graph theoretic approach. Stone has proposed an approach based on the network flow problem using the maximum flow algorithm developed by Ford and Fulkerson [59]. This approach forms a basis for later work in this area. The network flow problem and its application to task assignment are presented below.

The maximum flow problem involves a commodity network graph which consists of source nodes, sink nodes and several interior nodes; interior nodes are neither sources or sinks. All nodes are linked by weighted branches; source nodes represent production centers, and sink nodes represent demand centers. The branches represent commodity transport linkages, with weight of a branch indicating the capacity of the corresponding link. The value of a commodity flow is the sum of the net flows out of the source nodes of the

network which equals the sum of the net flows into the sink nodes. The maximum flow in the network is obtained by finding the minimum cutset. A cutset of a commodity network graph is the set of edges which when removed disconnects the source nodes from the sink nodes. The weight of the cutset is equal to the sum of the capacities of the branches in the cutset. The weight of the minimum cutset gives the maximum flow in the commodity network.

Using the network flow model described above, a system consisting of k processors and n tasks can be modeled as a network in which each processor is a distinguished node (source/sink) and each task is an ordinary node (interior node). An edge between pairs of tasks t_i and t_j with weight c_{ij} represents the communication costs between two tasks. As shown in Figure 3.6, an edge is drawn from each task node t_i to each processor node p_q with the weight

$$w_{iq} = \frac{1}{k-1}\sum_{r \neq q} x_{ir} - \frac{k-2}{k-1} x_{iq}, \tag{3.13}$$

where x_{ir} and x_{iq} represent costs of execution of task t_i on processor p_r and p_q respectively. A k-way cut in this network can be defined to be the set of edges that partition the nodes of the network into k disjoint subsets with exactly one processor node in each subset. Each subset represents assignment of tasks to processors. The cost of a k-way cut is defined as the sum of weights of the edges in the cut. The cost of the k-way cut exactly equals the total sum of execution and communication costs incurred by the assignment.

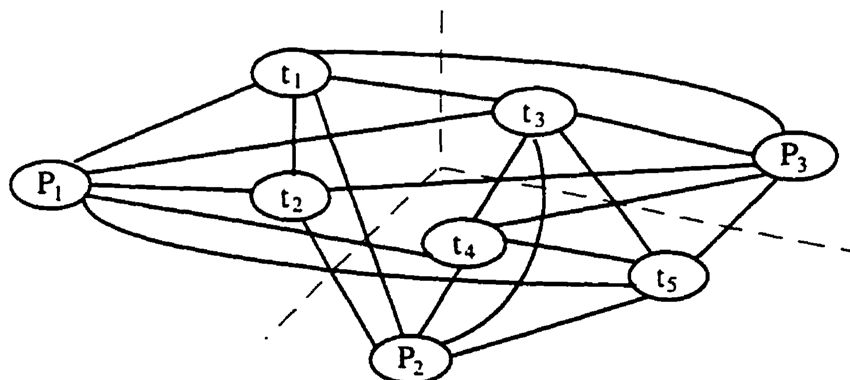


Figure 3.5: A n -processor network.

As discussed above, in a two processor system, a maximum flow corresponds to a minimum cut. Therefore an optimal assignment of tasks to processors in a two processor system can be obtained by using a *maximum flow/minimum cut* algorithm. The running time of the algorithm can be bounded above by the fifth power of the number of nodes in the graph. However the problem is NP-hard for arbitrary k. In addition, this approach does not make any effort to exploit concurrency. Virginia Mary Lo has proposed a family of heuristic algorithms [60] to extend Stone's approach. Her algorithms yield assignments with a greater degree of concurrency.

The integer or mathematical programming approach is based on an implicit enumeration algorithm subject to some additional constraints. Enumeration involves exploring every possible assignment of tasks to processors. The number of combinations that must be enumerated given n tasks and k processors is k^n. The method is implicit because some combinations can be eliminated without being fully explored, due to the presence of constrains. Ma et al. [58] have proposed an approach based on branch-and-bound. It is described below.

The task allocation method developed by Ma et al. [58] consists of three steps. The first step is to compute a cost function based on inter-processor communication and processing cost. Then a set of constraints to meet the requirements of the application are formulated and an algorithm is used to obtain the minimum total cost solution. Important constraints included in the model are task preference and task exclusion. Task preference is specified by a matrix P. If $P_{ij} = 0$ then task i cannot be assigned to processor j. Similarly exclusion is specified using matrix E. If $E_{ik} = 1$, then task i and k cannot be assigned to the same processor.

The branch-and-bound based approach proposed by Ma et al. [58] represents the partitioning/allocation problem as a search tree. The number of levels in the tree correspond to the number of tasks. The allocation decision represents a branching at the tree node corresponding to the given task. A two-level search tree with two processors is shown in Figure 3.7. The Branch-and-bound technique used by Ma et al. employs nine rules, (B, S, D, F, L, U, E, BR, RB), to *prune the tree* and thus reduce the search space. A few of the rules are presented below to illustrate the idea.

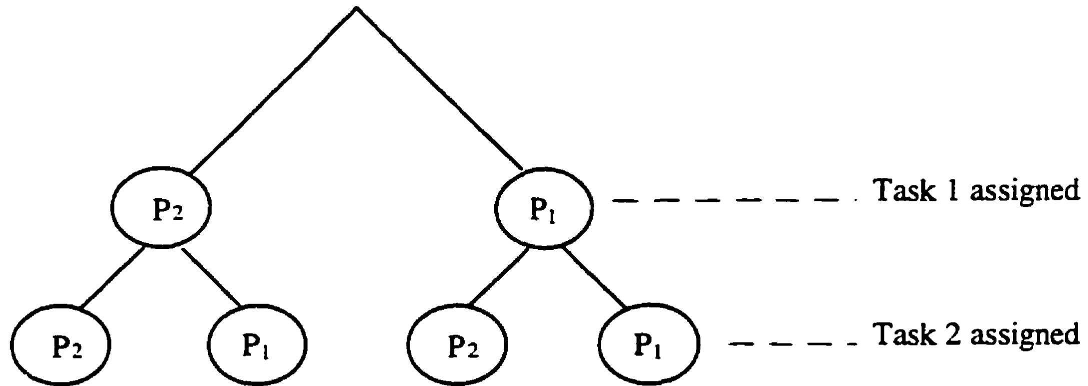


Figure 3.7: A two-level search tree with two processors.

The following rules determine whether the selected branch for a given node *k* should be eliminated. The Rule F checks the preference matrix for task *k* and processor *i*. If $P_{ik} = 0$, then the branch is eliminated. Similarly rule E checks the exclusion matrix. The Rule RB checks the cumulative memory requirement of tasks against the processor memory capacity. Rule D compares the partial cost L with the complete cost U. If L is greater than U, then the solution cannot be improved. Hence branch *i* for node *k* is eliminated. In addition to the elimination rules mentioned above, the following rules are applied to select the next node to investigate and to terminate the algorithm. The selection rule S selects the next node to be expanded. The branching rule B selects the processor allocation for a given node. The terminating rule BR terminates the algorithm when all possible paths have been investigated or a pre-specified number of iterations has been reached.

The integer programming technique seems best suited to applications where the goals of the allocation are complex and numerous. If more constraints are placed on the allocation possibilities, the search space reduces. Theoretically the algorithm proposed by Ma et al. [58] generates an optimal allocation with respect to inter-processor communication costs. However for large problems (many processors and/or tasks) achieving optimality remains intractable. It is suggested that the algorithm can be stopped after a certain

number of iterations, although this may not provide an acceptable solution. Since the tree is searched depth first, the paths on the right hand side of the tree may be left totally unexplored.

As mentioned above, the problem of finding optimal assignment of tasks to processors is NP-hard except in very restricted cases [42]. Therefore research has focused on development of heuristic algorithms to find sub-optimal assignments. Several algorithms use a classical graph theoretic approach or a mathematical programming approach to formulate the problem, and appropriate heuristic techniques are used to obtain a sub-optimal solution. Researchers have also proposed application of Simulated Annealing [61] and genetic algorithms [62] to solve the partitioning and assignment problem. A wide variety of heuristic techniques are described in the literature. A few of these approaches are briefly described below as examples.

Virginia Mary Lo [60] has proposed a family of heuristic algorithms to extend Stone's graph theoretic approach. Her heuristic repeatedly uses a max-flow min-cut algorithm to find optimal mapping. It consists of three parts: *Grab, Lump, and Greedy*. In *Grab*, the n processor network is converted to a two processor network consisting of a selected processor and a super-node which represents the other *(n-1)* processors. A *maximum flow/minimum cut* algorithm is then applied to this two processor network to find those tasks that would be assigned to the selected processor. This procedure is repeated for all processors. *Grab* may yield partial assignment of tasks to processors; if *Grab* halts with unassigned tasks, then *Lump* is invoked. *Lump* uses computation times and inter-task communication costs of unassigned tasks to test the possibility of assigning them to one processor. *Greedy* locates clusters of un-lumped tasks with high inter-task communication costs. These are assigned to the same processor.

A wide variety of clustering schemes are presented in the literature. Clustering schemes used in [50] and [63] reduce the number of tasks by forming task clusters. Tasks with maximum data exchange are merged to form a task cluster. These task clusters are assigned to processors. Finally to balance the load on processors, modules are shifted from heavily loaded processors to lightly loaded processors. Chen and Eshaghian [64] have proposed a clustering scheme with time complexity of $O(MN)$ where M is the

number of tasks and N is the number of processors. They have compared their approach with other leading techniques and shown that their mappings are similar or better, require less computing time and fewer or an equal number of processors. Clustering schemes are usually simple to implement and fast, however no attempt is made to enumerate all possible partitions; they tend to find a local minimum. An optimal or near optimal solution is not guaranteed.

The early research done on graph theoretic approaches formed a basis for later work on static partitioning. This approach defined the problem and several heuristics schemes for its solution were later developed. The graph theoretic approach, however, is mainly of theoretical importance. Its application to practical systems consisting of several processors and tasks is difficult, for example, Stone's approach [54, 55] is NP-hard for an arbitrary number of processors. The integer programming technique is best suited to applications where the goals of the allocation are complex and numerous. If more constraints are placed on the allocation possibilities, the search space reduces. Theoretically, this approach provides an optimal solution, however for large problems (many processors and/or tasks) achieving optimality remains intractable. Limitations of graph theoretic and integer programming approaches have led to development of a wide variety of heuristic techniques. These schemes do not guarantee optimal or near optimal solutions, however heuristic techniques are commonly used due to their simplicity and low computational cost. A scheme based on heuristic solution of bin packing problem is used in this thesis. A brief description of the scheme and reasons for its use are presented in Chapter 4.

3.8 Placement Techniques

Several researchers have divided the static partitioning problem into two parts: partitioning the task graph into a number of partitions less than or equal to the number of processors, and assignment of the partitions to processors. Placement is the assignment of partitions to processors, although the placement problem is also known as a mapping problem. With n processors and partitions, $n!$ distinct placements are possible. The objective

of a placement algorithm is to select a partition with minimum inter-processor communication cost.

The placement problem can also be viewed as a graph mapping problem [65, 66]. In this case, a task graph is matched against the system graph in order to minimize the inter-processor communication cost. This problem is also equivalent to the graph isomorpism problem which is known as a classically difficult combinatorial problem. Several heuristic placement algorithms are presented in the literature. Most of these are developed for a specific application domain and a parallel architecture. An approach proposed by Bokhari [65] is briefly described below.

Bokhari has described his approach using an example of structural analysis problem solved using a finite element machine (FEM), an array of processors developed at NASA Langley Research Center. He assumes that all edges of the task graph have equal weight and number of tasks/task clusters are less than or equal to the number of processors. The quality of mapping is determined by the number of problem edges that fall on array edges. This number is called the cardinality of the mapping. The algorithm starts with the adjacency matrix of the task graph, and the adjacency matrix of a square FEM onto which it is to be mapped. A permutation of the task graph adjacency matrix that matches more closely with adjacency matrix of the FEM is produced as the output.

The algorithm starts by examining the pair-wise exchange of each node with every other node. The pair-wise exchange with the largest gain in cardinality of mapping is examined. If the gain is greater than or equal to zero then that exchange is made. The pair-wise exchange process is stopped if no exchange leads to an improvement. The best mapping obtained by the pair-wise exchange heuristics is stored. The pair-wise exchange heuristics is not guaranteed to provide the best mapping. Some mappings are not optimal and can not be improved by a pair-wise exchange. Bokhari refers to these mappings as *dead ends*. The algorithm attempts to leave the *dead ends* by randomly exchanging n pairs of nodes. The pair-wise exchange process is then resumed. If the mapping obtained after the random exchange is poorer than the previously obtained best mapping then the algorithm terminates.

Bokhari's approach has several limitations. He assumes that all edges of the task graph have equal weight and number of tasks/task clusters are less than or equal to the number of processors. Therefore his approach is useful for only a selected class of problems. However several researchers have enhanced. Bokhari's approach [65], for example, Lee and Aggarwal's approach attempts to remedy several limitations of Bokhari's approach. It is explained below.

Lee and Aggarwal's approach [66] also assumes that the number of task graph nodes is not greater than the number of processors, however task graph edges are allowed to have different weights. Their approach involves accurate characterization of the communication overhead and it is assumed that the communication in parallel systems takes place in phases. A *phase* is the time interval during which the communication for a problem edge is carried out. It is observed that communication along certain problem edges is required in the same phase, and in some others, in different phases. Some problem edges are used more frequently than others indicating that greater weight has to be given to that edge. In addition, a system link may be shared by multiple problem edges communicating in the same phase which may change the communication overhead of the corresponding problem edges.

Lee and Aggarwal [66] formally express communication overhead in terms of objective functions *OF1*, *OF2* and *OF3*. The objective function *OF1* assumes that no two problem edges are required in the same phase, and *OF2* assumes that all problem edges are required in the same phase. The objective function *OF3* assumes a combination of the previous two cases. An appropriate selection of the objective function depends on the needs of the application.

Lee and Aggarwal's [66] algorithm consists of two parts: initial assignment, and a pair-wise exchange. In the assignment phase, a task with the largest communication intensity is selected and assigned to a processor with degree (the number of communication links) as close as possible to the task. Then a task which is adjacent to already assigned tasks and has the highest communication intensity is selected. It is placed on a processor such that certain measure derived from a selected OF is minimized. This process is used for placement of all tasks. The pair-wise exchange part of the algorithm

attempts to improve the initial mapping. Lee and Aggarwal's approach does not exhaustively exchange all pairs of tasks. The candidate task is selected according to a measure derived from the selected OF. An exchange of this task with all other tasks is attempted and the exchange giving the smallest OF is accepted. The time complexity of Lee and Aggrawal's algorithm is $O(n^3)$ where n is the number of tasks and processors.

To reduce the complexity of the placement problem, a number of approaches such as graph contraction and clustering have been studied [50, 67, 68]. Most of these graph matching based techniques only cluster the task graph. The clustered task graph is then matched with the system graph. Most of these techniques perform partitioning and allocation. Chen and Eshagian [64] have proposed a fast recursive placement algorithm. It clusters both system and task graphs into a hierarchy of clusters. Clustering of task and system graphs is done only once, independent of one another. The highest level task clusters are first mapped on to the system cluster. Then the mapping is done recursively at each clustering level. Their algorithm can be used for directed and undirected task graphs. The time complexity of their algorithm is $O(mn)$, where m is the number of tasks and n is the number of processors.

3.9 Summary

This chapter describes issues involved in implementing circuit simulation on parallel computers. Shared memory and distributed memory parallel architectures are introduced. Shared memory architectures use a global, shared memory for inter-processor communication and coordination, and distributed memory architectures connect multiple autonomous processing modules using a processor-to-processor interconnection network. Shared memory programming model is similar to the uniprocessor programming model, therefore these machines are easier to program. Shared memory architectures are, however, complex and expensive and synchronization for shared resources limits their scalability. Distributed memory computers are relatively less expensive and scale well, however it is usually difficult to program distributed memory computers.

The performance of parallel applications is limited due to *sequential fraction*, *communication overhead* and *load imbalance*. It is observed that an algorithm that causes some extra work can still be useful for implementation on distributed memory computers if it has low sequential fraction. This is mainly due the low cost of adding extra processors.

The rest of the chapter is devoted to the analysis of parallelism in circuit simulation. Relaxation-based simulators often use direct methods for simulation of sub-circuits; therefore parallelism in direct methods is studied. Issues involved in implementing parallel relaxation based simulators using shared and distributed memory computers are analyzed. Waveform relaxation techniques have two forms of parallelism: coarse grain parallelism across sub-circuits and fine grain parallelism within a single Newton-Raphson iteration of a sub-circuit. Techniques used to exploit both forms of parallelism are described.

The circuit simulation program presented in this thesis uses a distributed memory computer. Appropriate partitioning and allocation of program segments is essential for efficient parallel implementation on distributed memory machines. Therefore partitioning and allocation issues are studied at the end of the chapter. The following chapter describes an implementation of a parallel waveform relaxation based simulator.

4. PARALLEL WAVEFORM RELAXATION IMPLEMENTATION

An objective of this thesis is to implement a parallel circuit simulation program in order to demonstrate and test the design concepts developed during this project. A typical circuit simulation program consists of a number of analysis modules each performing a different kind of analyses, for example ac, dc or time-domain transient analysis. This chapter concentrates on the parallel implementation of a transient analysis module. Parallel implementations of Gauss-Seidel and Gauss-Jacobi parallel waveform relaxation programs for distributed memory machine are described.

Waveform relaxation programs are implemented using a Transtech MCP1000 parallel processing board [69] which is based on the INMOS *IMS T800* Transputer [20]. Transputers are a family of microprocessors specifically designed for parallel processing. The *IMS T800* Transputer has four high-speed serial communication links which can be used to create many different network topologies. Transputers are attractive due to their low cost to performance ratio. In addition, MCP1000 boards plug into a slot of standard workstations. This would facilitate development of low cost multiple processor systems on workstations for VLSI CAD applications.

A distributed memory parallel programming language, *Occam2,* specifically developed for Transputers, was used for the implementation [70]. The basic principle of *Occam* is simplicity; unnecessary duplication of language mechanisms is systematically avoided. *Occam*, unlike parallel versions of sequential programming languages, was designed to support concurrency. It is based on C. A. R. Hoare's theoretical model of *Communicating Sequential Processes (CSP)* [71]. Within the *CSP* framework, each program is a collection of sequential processes, each of which may be executing concurrently with others. The processes interact or communicate only via

synchronized input/output operations. The use of the CSP model simplifies the task of program verification, by allowing application of mathematical proof techniques to prove correctness of programs. In addition, the formal semantics of the language facilitates automatic synthesis of *Occam* programs from high-level specifications. Appendix C gives a brief description of Transputers and *Occam2*.

The chapter consists of six sections. Possible strategies used to implement waveform relaxation programs on distributed memory machines are described and compared in Section 4.1. A distributed queue approach to exploit coarse grain parallelism across sub-circuits has been selected for implementation. The overall structure of the parallel waveform relaxation program using the distributed queue approach is described in Section 4.2. The program consists of three modules: input processing, task graph partitioning, and parallel transient analysis. The input processing module is described in Section 4.2. Algorithms used to partition the sub-circuit task graph into a number of partitions equal to the number of processors are presented in Section 4.3. Section 4.4 describes the algorithm used for placement of partitions on processors. The parallel transient analysis module is described in Section 4.5 and Section 4.6 presents a summary of the chapter.

4.1 Implementation Strategies

The organization of algorithms for parallel execution has been approached in a number of different ways. These approaches can be broadly classified into three parallelism categories: *algorithmic*, *geometric* and *process farming* [61]. Algorithmic parallelism has quasi-independent tasks which execute sections of the algorithm. Tasks are usually non-identical and data and computed results are passed among the tasks. A pipeline is an example of algorithmic parallelism. In geometric parallelism tasks are quasi-independent and identical; each task operates on a part of the data and interacts with the other tasks according to the geometry of the problem. Geometric parallelism can be used for solving, for example, finite element analysis problems. In process farming, tasks are independent but identical and data are processed

in a random order. A typical application area for process farming is image processing.

In the context of parallel waveform relaxation, a sub-circuit solution task can use the process farming approach for evaluating device models. The application of process farming approach for parallel model evaluation is described in the following sub-section. Sub-circuit evaluation tasks are quasi-independent and identical, they interact with one another according to the circuit topology. Therefore the parallelism across sub-circuits can be classified as geometric. Techniques to exploit parallelism across sub-circuits are described in Sections 4.1.2 and 4.1.3.

4.1.1 Parallel Model Evaluation

Section 3.3 describes the use of the process farming approach to exploit fine-grained parallelism in the *FORM* phase of the parallel direct method. Issues involved in the use of process farming based parallel model evaluation for waveform relaxation are similar to parallel direct methods. A typical implementation consists of a queue of sub-circuits on a processor called the root processor. Most computations associated with the global waveform relaxation implementation (for example, window selection, time-step control, linear equation solution and convergence checks) are performed by the root processor and device model evaluation tasks are dynamically assigned to free processors. This approach is conceptually simple; however it is efficient only for a small number of processors. It has been observed that if the number of processors is more than three or four then speed-up saturates because additional processors remain idle for most of the time. This serves as a motivation for exploiting coarse grain parallelism across sub-circuits. Two approaches, a single queue and a distributed queue, can be used for exploiting geometric parallelism across sub-circuits. These are described below.

4.1.2 Single Queue Approach

The single queue approach is similar to the process farming approach. In this approach a central queue of sub-circuits is maintained on the root

processor. Sub-circuits are simulated using a data-flow driven scheduling mechanism. A circuit ready for simulation is sent to a free processor. After simulating the circuit the processor returns the results to the root processor. Since all computations associated with sub-circuits are performed on worker processors, the amount of sequential computation performed on the root processor is low. The root mainly performs window selection and scheduling. An important advantage of this technique is its ability to perform dynamic load balancing due to run-time assignment of sub-circuits to free processors.

The applicability of this approach is mainly limited due to high communication costs. It is necessary to send all node voltage waveforms, initial conditions, window boundaries, and the sub-circuit netlist to worker processors. Worker processors return the node voltage waveforms, and convergence information to the root. Transputers provide hardware support for communication. Therefore a goal of most Transputers based implementations is to overlap computations with communications so that message passing latency can be hidden from the applications. In the context of a single queue approach, it is desirable to ensure that the sum of total message passing latency and message establishment costs for a sub-circuit is less than the cost of computing a window iteration for the sub-circuit since the message passing latency costs can affect utilization of processors. The mean distance between the root processor and a worker processor depends on the network topology and the number of processors. In most practical topologies, it increases with an increase in the number of processors. Therefore the mean communication cost also grows with the number of processors. In addition, the root processor and communication links near the root become a bottleneck. Another disadvantage of this approach is poor memory utilization.

4.1.3 Distributed Queue Approach

The distributed queue approach statically partitions the sub-circuit task graph into partitions equal to the number of processors; each partition is statically assigned to a worker processor. The root processor is responsible for implementing a barrier synchronization point to ensure that all sub-circuits have completed a window iteration before the next iteration is

started. The root also computes the size of the next window. The amount of sequential computation performed on the root is very low. Worker processors examine dependencies of sub-circuits (sub-circuit scheduling), simulate eligible sub-circuits and communicate node voltage waveforms to fanout sub-circuits. A worker completes a window iteration for each sub-circuit assigned to it and sends synchronization messages to the root. Thus this approach allows implementation of a distributed sub-circuit scheduling mechanism.

In most practical situations, communication of node voltage waveforms can be overlapped with the computation (with the exception of the last sub-circuit in the queue). It is also possible to combine synchronization messages to the root in order to avoid delays at the communication links near the root. This arrangement has less communication overhead as compared with the single queue approach because sub-circuit description and initial values are communicated to a worker processor only once during a simulation interval. The inter-processor (inter-worker) communication which consists of messages for updating the node voltage information for fanout circuits can be effectively overlapped with computation.

This approach statically partitions the sub-circuit task graph. Circuit simulation problem exhibits highly data dependent behavior, therefore *a priori* estimation of task sizes is difficult. In addition, task sizes vary during a simulation interval. These factors make effective static partitioning difficult, so load imbalance can become a dominating source of overheads. Each approach has its advantages and limitations but the distributed queue scheme is potentially more scalable. Therefore it has been used in this thesis.

4.1.4 Multi-computer Interconnection Network

The MCP 1000 parallel processing board provides software support for configuring the multi-computer interconnection network topology. The multi-computer interconnection topology used for the parallel implementation is shown in Figure 4.1. This topology can be easily realized using *4* links of *IMS T800* Transputers. The following sections describe an implementation based on a distributed queue scheme.

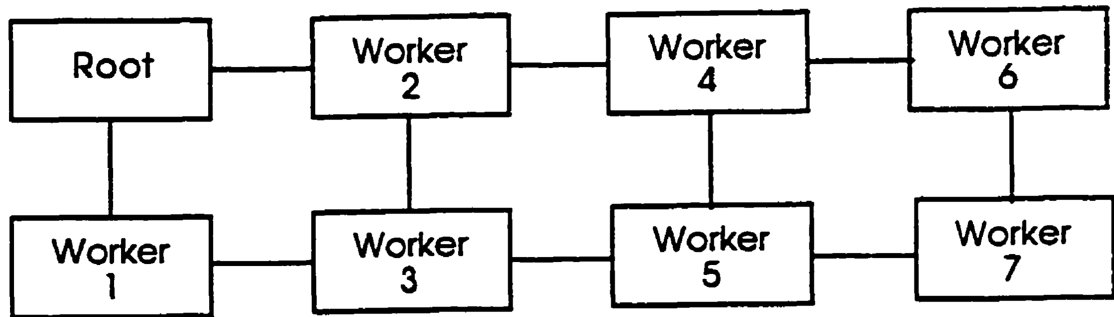


Figure 4.1 Network Topology

4.2 Program Structure Chart

A program structure chart for the parallel circuit simulation program is shown in Figure 4.2. The circuit simulation program is divided into three modules: input processing, task graph partitioner and parallel transient analysis. The input processing module reads the circuit description written in a hardware description language. It consists of program segments for parsing (syntax checking), filling intermediate data structures, DC analysis, and circuit partitioning.

RELAX2.3 is a sequential program developed at the University of California, Berkeley [14]. The input processing module of RELAX2.3 was modified and adapted for this application. The circuit partitioning program segment uses a Norton equivalent conductance partitioning algorithm to partition a given circuit into a number of sub-circuits. This algorithm is described in Section 2.5.2. The input processing module also generates information about dependency relationships among sub-circuits.

The task graph partitioning module accepts this information and generates a sub-circuit task graph. The Gauss Seidel algorithm treats the sub-circuit task graph as a directed graph and the Gauss Jacobi algorithm considers it as an undirected graph. Therefore, appropriate selection of a partitioning

algorithm depends on the type of WR algorithm to be used. Task graph partitioning algorithms used in this thesis are described in Section 4.3.

Each partition is loaded on an independent processor for transient analysis using the placement algorithm described in Section 4.4. Relationships among circuit partitioning, task graph partitioning, and placement stages are shown in Figure 4.3. The input processing and task graph partitioning modules are difficult to convert to parallel form; however these modules require only a fraction of the total simulation time. Therefore these modules are executed sequentially on the host workstation.

The parallel transient analysis module can be divided into two parts: a transient analysis application and a process framework. The transient analysis application consists of program segments for performing sub-circuit scheduling, window selection, time step control, device model evaluation, linear equation solution, and convergence check.

The sub-circuit scheduler checks all dependencies of a sub-circuit before simulation begins. Window selection and time step control algorithms are discussed in Chapter 2 while device model evaluation techniques are described in Appendix A. The program has device models for resistors, capacitors, diodes and MOS transistors. The waveform convergence checker is used to compare waveforms from the present iteration with the previous iteration.

The process framework consists of program segments for implementing message routing and buffering; it also provides run-time support to the parallel application. The process framework is described in Section 4.5. The parallel transient analysis module consists of approximately *5000* lines of *Occam2* code. The following section describes algorithms used by the task graph partitioning module.

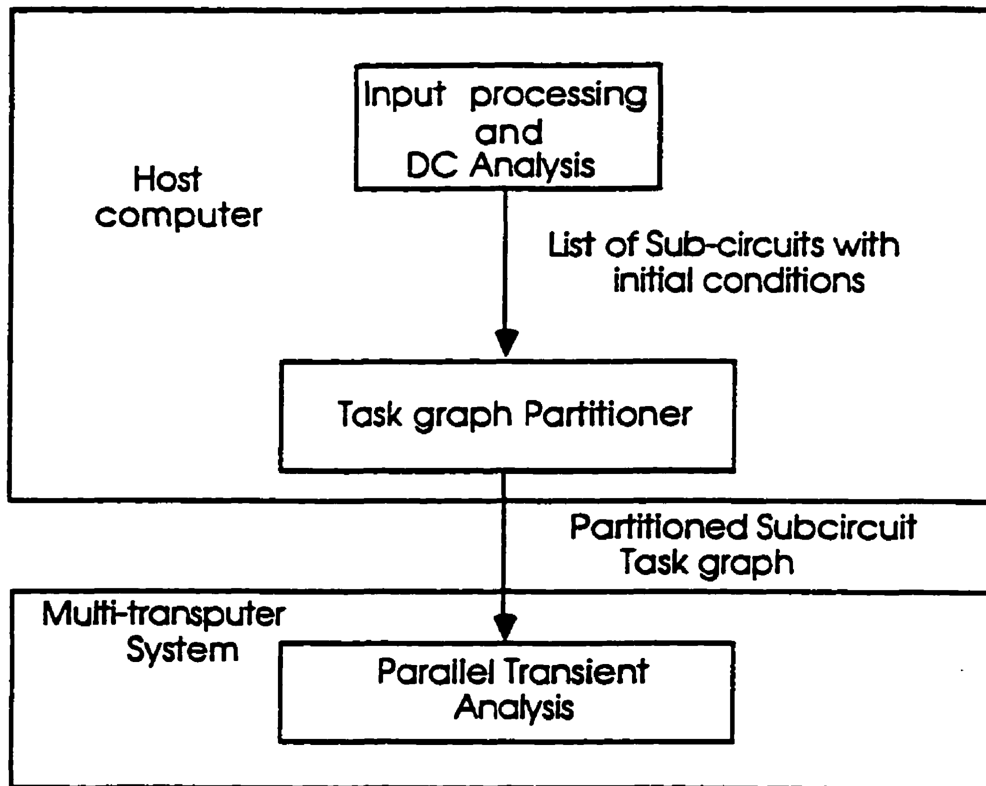


Figure 4.2: Program structure chart.

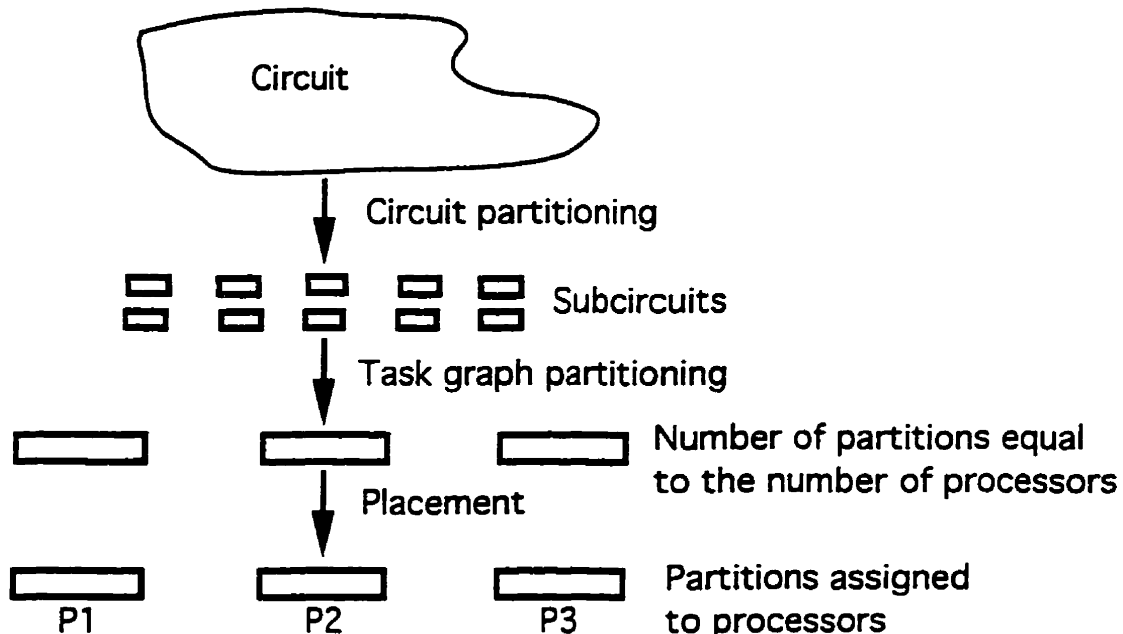


Figure 4.3: Circuit partitioning, task graph partitioning and placement.

4.3 Partitioning Algorithms

As mentioned in the previous section, the input processing module generates a sub-circuit task graph. This task graph is partitioned into a number of equal partitions and each partition is allocated to a processor. It is known that partitioning and allocation problems are NP-complete, hence heuristic techniques are commonly used. Newton et al. [3] have reported that for small circuits, i.e. for circuits with less than 20 nodes, device model evaluation time dominates the matrix solution time, so sub-circuits are usually small. Therefore the partitioning algorithm assumes that a sub-circuit simulation task size (node weight) is proportional to the number of devices in the sub-circuit. Most of the communication is overlapped with computation, and message establishment costs are very small compared with a sub-circuit simulation time. Therefore the partitioning algorithms attempt to minimize load imbalance overhead. The allocation algorithm has been developed to minimize communication costs.

An algorithm based on the Critical Path Method (CPM) [72] has been used to partition the directed task graph generated by the GS WR algorithm. This technique has been selected for the present system because the GS WR task graph does not have any regular structure and CPM based techniques are established methods for scheduling unstructured directed task graphs on parallel computers. An algorithm based on the solution of the bin packing problem has been used for partitioning the undirected GJ task graph because it is a simple approach. Since exact task sizes are not known *a priori*, a complex technique is unnecessary. These algorithms are described below.

4.3.1 Partitioning Algorithm for Gauss-Seidel Task Graph

The CPM algorithm is based on two concepts: tasks on the critical path have to be executed in sequence, and the sum of execution times of tasks on the critical path determines the minimum execution time of a program. The algorithm ranks nodes according to the lengths of their critical paths and generates a priority list. Then the node with the highest priority is assigned to the most appropriate PE. The following definitions are used for describing the CPM algorithm. [72]

DEFINITION 1: *Accumulated time* of a processor PE_i, $AT(PE_i)$ is the total time required for PE_i to finish all tasks assigned to it.

DEFINITION 2: A task graph node is said to be *mature* at time t_a if it has no parent nodes or if all of its parent nodes are already assigned to some processor PE_k and $AT(PE_k) \leq t_a$ for all k.

DEFINITION 3: An *exit path* of a task graph node P is the longest path from P to a leaf node of the DAG. If there are many maximal paths from a node then any one of them can be an *exit path*. It is important to note that finding a length of the critical path is equivalent to finding an *exit path* for a node.

The first step in the CPM algorithm is to determine the length of an exit path for each node. The exit path algorithm is described below. Let n be the number of nodes in the DAG. Let N_i and L_i be the out degree and the length of exit path for node P_i. Let CL be a list. Initially CL is empty , $L_i = 0$; for all i. and $N_i = 0$; for all leaf nodes.

Exit Path Algorithm:

1.0 For all leaf nodes P_k assign the weight of P_k to L_k. Add all leaf nodes to CL.

2.0 Let P_i be a node in CL. Repeat steps 2.1-2.3 until CL is empty:

2.1 For each parent node P_m of node P_i do:

2.1.1 $N_m = N_m - 1$

2.1.2 If $L_m < WT(P_m) + L_i$, then $L_m = WT(P_m) + L_i$.

2.2 If $Nm = 0$ and P_m is not an entry node, add P_m to CL.

2.3 Remove P_i from CL.

The exit path algorithm calculates exit path lengths for all DAG nodes. In this algorithm, each node is processed only once in step 2.0 and steps 2.1-2.3 can be repeated a maximum of n times. Therefore the time complexity of the algorithm is $O(n^2)$.

The CPM algorithm:

1.0 Call exit path algorithm

2.0 Repeat steps 2.1-2.4 until the accumulated time for all the processors is the same and there are no mature nodes available.

2.1 Choose processor PE_i such that $AT(PE_i)$ is the smallest.

2.2 Find the mature node with the largest exit path length.

2.3 If found assign it to PE_i.

2.4 If there are no mature nodes available, then assign a dummy node to processor PE_i such that $WT(P_{dummy}) = AT(PE_m) - AT(PE_i)$ where $AT(PE_m) > AT(PE_i)$ and $WT(P_{dummy})$ is the weight of the dummy node.

log 2

The information regarding the length of an exit path is kept in a heap. Step 1 requires $O(n^2)$ iterations. Step 2 is executed n times, and Step 2.2 requires $O(\log n)$ time to complete. Therefore the time complexity of the CPM algorithm is about $(n\log n + n^2)$ which is, $O(n^2)$.

4.3.2 Partitioning Algorithm for Gauss-Jacobi Task Graph

Since the task graph generated by the GJ WR algorithm is undirected, an algorithm based on the solution of the bin packing problem is used for partitioning the task graph. The algorithm is similar to a *non increasing best fit heuristic*. It is described below.

1. Arrange n sub-circuit tasks in a non increasing (descending) order of weights.

2. Assign first m tasks to m processors in any suitable order.

3 For remaining (n - m) tasks, repeat steps 3.1 and 3.2

3.1 Determine the processor with the lowest accumulated weight. In the case of conflict, a select processor with lowest number of tasks.

3.2 Assign the next largest sub-circuit task to this processor.

Since the number of tasks is usually greater than the number of processors, the order of complexity of the algorithm is determined by step 1 which is $O(n\log n)$. The two algorithms discussed above partition the given task graph into partitions equal to the number of processors. It is necessary to assign a partition to a processor using an algorithm as described below.

4.4 Placement Algorithm

The allocation algorithm assigns partitions to worker processors in such a way that communication cost is minimized. The cost of communication between two partitions is proportional to the number of DAG edges shared by the two partitions. Each edge is assigned a unit weight The following algorithm is used.

1.0 Select a partition pair with the highest communication and place it on two adjacent processors.

2.0 Select a partition with highest communication with any one of the placed partitions and place it on an adjacent processor.

3.0 Repeat Step 2 until all partitions are assigned.

Each worker runs a copy of the parallel transient analysis module. The organization of the parallel simulation program and the implementation of the transient analysis module are described in the following section.

4.5 Parallel Transient Analysis

The eight processors used for the application are organized as one root Transputer and seven worker Transputers. The root Transputer reads sub-circuits from the host's file system and loads them on appropriate processors. It also synchronizes the operation of the worker Transputers. The root Transputer initiates simulation by sending the size of a window to worker processors which evaluate a window iteration for all sub-circuits assigned to them and communicate appropriate waveforms to fanout sub-circuits residing on other processors. Worker processors also compare waveforms obtained during the present iteration with those obtained from the previous iteration and determine how far the waveform convergence has progressed. Workers communicate two values: the maximum number of time points necessary for describing waveforms, and the smallest value of the waveform convergence point to the root. When all waveforms converge at the end of the window, the root calculates the size of the next window depending upon the maximum number of time points and iterations required for the previous window and communicates it to the workers.

The main data structures used for the implementation are shown in Figure 4.4. *Occam 2* does not provide pointers, or facilities for dynamic memory allocation and composite data types such as *C* structures or *Pascal* records. Therefore single and multi-dimensional arrays are used for implementing data structures. A row of the *Sub-circuit list* array provides all information about a sub-circuit assigned to a processor. Information about all devices is stored in a single vector and the *Sub-circuit queue* along with *Ready* and *Simulated* (both arrays of flags) are used for sub-circuit scheduling. All circuit node voltage waveforms on a processor are stored in voltage and time arrays. Two instances of these arrays are used for storing waveforms from the present and previous iterations. Iteration counts of waveforms are used for scheduling waveforms for execution. A *Sub-circuit fanout list,* and a *Subcircuits_to_processors map* are used for sending messages to fanout sub-circuits residing on other processors. A transient analysis application module running on a worker processor is embedded in a process framework as described below.

4.5.1 Process Framework

The process framework provides a run-time environment for the application. It consists of program segments for message routing, buffering and synchronization. The process framework has been designed to improve utilization of links/processors, minimize communication latency, and avoid deadlocks. Transputers provide hardware support for inter-processor communication using autonomous DMA engines. Therefore Transputer links can perform bi-directional data transfers without seriously affecting the processor performance [73]. It is possible to improve utilization of links and processors by de-coupling computation from communication. Independent concurrent *Occam* processes are used for communication and computation. Communication latency is minimized by appropriate design of process configuration and the use of an efficient message routing algorithm. Occam processes communicate using synchronous messages. This messaging paradigm is prone to deadlocks. Communication deadlock avoidance techniques typically involve use of intermediate buffer processes which

allows asynchronous messaging among communicating processes. A process framework designed for the present system is described below.

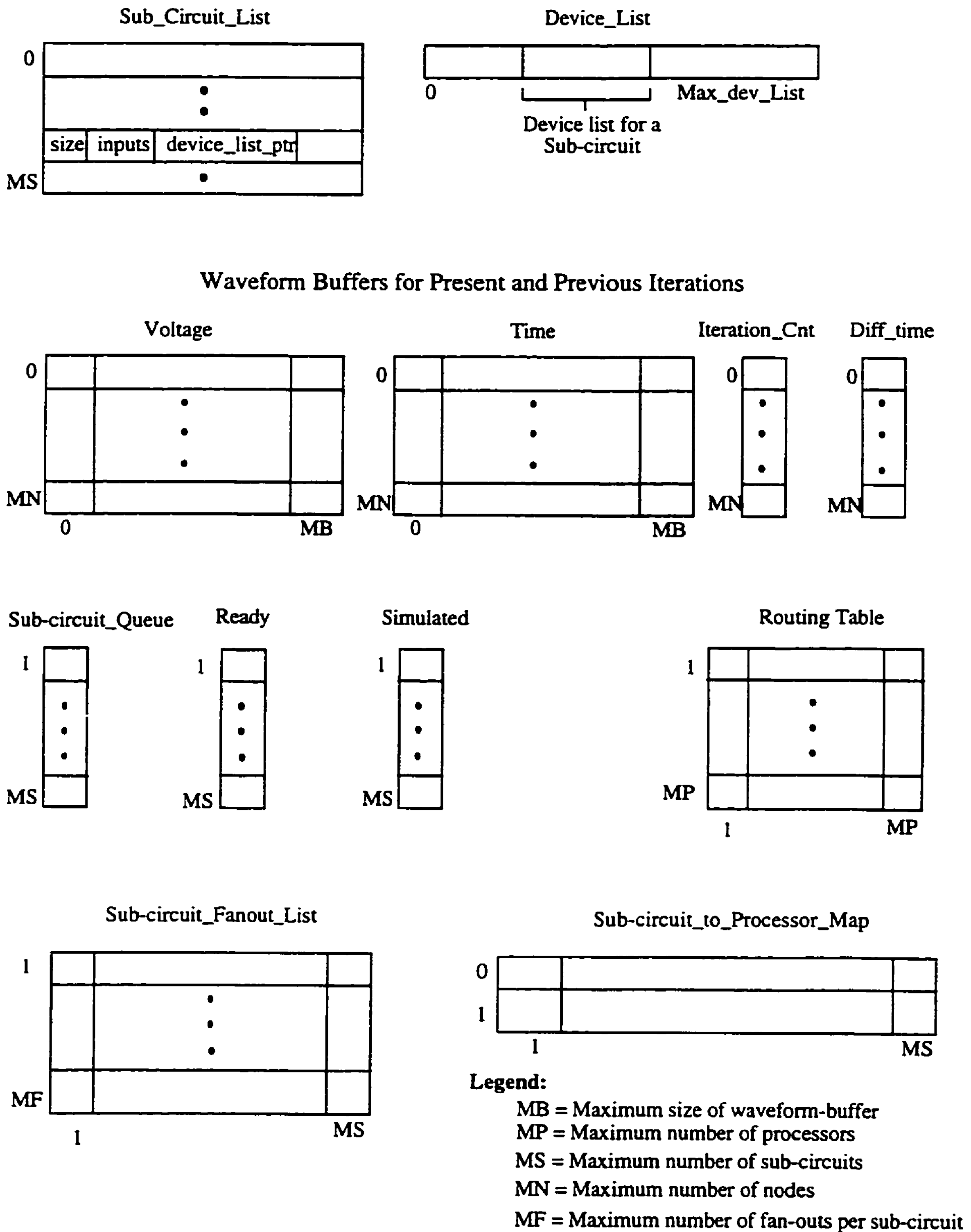


Figure 4.4: Data Structures.

Figure 4.5 shows a process framework running on a worker Transputer. Circles denote processes and arcs denote communication channels. The process framework consists of six router processes, a buffer process and an application process. Two router processes are associated with each link, one for input and the other for output. Router processes run in parallel with the application process and with one another which allows communication to be overlapped with computation. All links can send and receive messages simultaneously to reduce communication latency and improve link utilization. Router processes implement a shortest path routing algorithm. Router processes have a higher priority over the buffer and the application process.

The buffer process has been introduced for buffering input messages to make router processes free for accepting and transmitting messages. The buffer process is also useful for avoiding deadlocks. A handshaking protocol has been used for communication between the buffer process and the application process. A buffer process initiates communication by sending buffered messages received from the root and/or other Transputers. No further messages are sent until an acknowledgment is received from the application process. The application process receives messages, simulates sub-circuits eligible for execution, sends node voltage waveforms to appropriate fanout sub-circuits, and then sends an acknowledgment to the buffer process. This arrangement is necessary for avoiding deadlocks.

4.5.2 Instrumentation for Performance Measurements

The *IMS T800* Transputer has two on-chip hardware timers one for each priority level [20]; these are used to perform timing measurements. The high priority timer is incremented every microsecond and the low priority timer is incremented every *64* microseconds. Occam provides an interface to these timers using special input-only channels of type TIMER. Transputer/Occam timers are cyclic; therefore modulo arithmetic is used for calculating time delays. The cycle time for high priority and low priority timers are 1.2 hours and 76 hours respectively. These timers are used for performance all of the measurements.

Long time

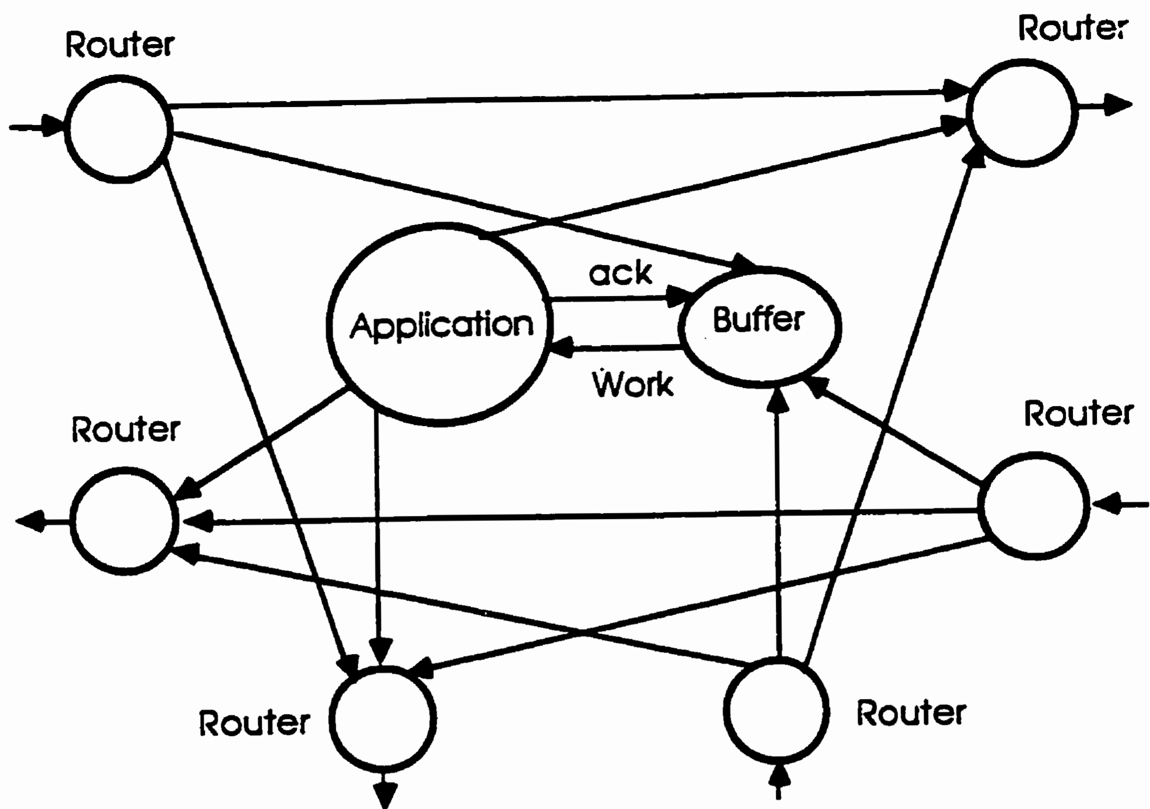


Figure 4.5: Process framework.

4.6 Summary

This chapter describes parallel implementations of waveform relaxation programs using a network of Transputers. Different strategies for parallel implementation such as, a single queue approach and a distributed queue approach are described and compared. The distributed queue approach is selected for exploiting the coarse grain parallelism across sub-circuits. This approach is more generic and scales well. A distributed queue implementation involves static partitioning and placement of sub-circuits on processors. The program uses an algorithm based on the critical path method to partition a GS task graph and a bin packing heuristics to partition a GJ task graph. These algorithms are described. In addition, a placement heuristic which attempts to minimize the communication overhead is also

discussed. The parallel processing framework provides a skeleton for implementing parallel applications. An efficient implementation of the processing framework which exploits characteristics of the underlying hardware is necessary to improve CPU utilization. Therefore, the parallel processing framework is discussed in detail. A dominating source of overhead in distributed queue implementation is load imbalance. The following chapter describes techniques to minimize this problem.

5. DYNAMIC LOAD BALANCING

Distributed memory multi-computers permit efficient implementation of large parallel applications, however, it is necessary to minimize the effects of performance limiting factors. Load imbalance, an uneven distribution of workload among processors due to which some processors are overloaded and other remain idle, is an important source of overhead in distributed memory machines. Load balancing techniques attempt to improve system performance by providing better utilization of resources in the entire system. The purpose of load balancing is to reduce the mean time to complete a job by distributing the workload evenly. Load balancing techniques are classified as either static or dynamic. Static techniques described in earlier chapters use *a priori* estimation of task sizes and communication overheads to partition and allocate tasks while, dynamic techniques perform load balancing during run time by migrating tasks from heavily loaded processors to lightly loaded processors.

Several load balancing schemes are described in the literature [52][74][75][76]. Most of these schemes are validated and compared using synthetic workload models and simulation, and very few practical implementations are presented. This chapter describes an implementation of a dynamic load balancing algorithm similar to the *contracting within neighborhood* approach [76].

The chapter consists of five sections. Limitations of static partitioning schemes are described in Section 5.1. Section 5.2 describes various dynamic load techniques. The load balancing algorithm used in this thesis is presented in Section 5.3 and its implementation is described in Section 5.4. Section 5.5 presents a summary of the chapter.

5.1 Limitations of Static Partitioning

Static partitioning schemes assume static task graphs, i.e. the number of tasks, task sizes and communication patterns among task sizes are fixed. However, in many application areas, for example, search problems and symbolic applications, tasks are dynamically created, also, applications developed using programming paradigms, such as functional and logic programming do not have static task graphs. This may result in load imbalance among processors and require dynamic load balancing.

Several scientific applications can be partitioned into a fixed number of tasks which do not change during run time. Although task graphs of these applications have fixed topology, task sizes may vary drastically during run time. Nicol and Reynolds Jr. have described a parallel solution of a system of partial differential equations arising in fluid dynamics [77]. In this problem, computation consists of several distinct phases wherein workload characteristics of each phase is different. Nicol and Reynolds Jr. have proposed dynamic re-mapping of computation after a phase change. Their approach is applicable to only a specific class of applications which have a few distinct phases.

Most numerical algorithms based on iterative methods show data dependent behavior. For example, the finest granularity task in the parallel solution of non-linear algebraic equations using non-linear relaxation involves an iterative solution of a single equation. The number of iterations required for convergence depends on the values of coefficients and initial values of iterates. In addition, *a priori* estimation of task, size which depends on the number of iterations, is difficult. Since some applications show a wide variation in inter-task communication patterns during run time. These applications may need dynamic load balancing to reduce the communication overhead.

The workload offered by a parallel waveform relaxation based circuit simulator depends on the nature of the circuit, complexities of device models, and input waveforms; it may vary widely during the simulation interval. The finest granularity task consists of computing a window iteration for a sub-circuit. Absolute and relative values of sub-circuit evaluation task sizes

vary widely due to input conditions, latency, partial waveform convergence and multi-rate behavior and this can be a major source of load imbalance. The cost of computing a window iteration for a sub-circuit depends on the number of time points used to describe the waveform and the cost of computing individual time points. Since an adaptive time step control is used, the number of time points necessary to describe a waveform vary across iterations. It is also difficult to predict the total number of time points. The number of Newton-Raphson iterations necessary to compute node voltage values at a time point depend on the spectral radius of the Jacobian matrix and initial conditions. Thus, sub-circuit evaluation task sizes vary widely over the simulation time interval. In addition, static partitioning is an NP-complete problem. Therefore only sub-optimal partitioning is practical or even possible.

5.2 Dynamic Load Balancing Methods

The nature of and reasons for load imbalance overhead depends on the application domain. A wide variety of techniques are proposed in the literature to meet the needs of different applications and no technique is universally applicable. Load balancing techniques can be classified according to control policy, information policy, initiation policy, transfer policy, and location policy [74].

Control policy refers to the agency making the load transfer decisions. Control policy can be centralized or distributed. Centralized schemes have a central controller that collects load information and makes load transfer decisions while in distributed schemes, each node is responsible for making transfer decisions and control authority is distributed. Centralized schemes can take near optimal load migration decisions, however, collection of state information from all nodes can be expensive. Distributed schemes usually perform local load balancing at a considerably lower cost.

Information policy determines the method of exchanging and using the load status information. Information policy can be static or probabilistic, where in a static or deterministic approach, current system state such as CPU utilization, memory utilization, and average response time are used to make

load migration decisions. In a probabilistic approach, an arriving job is sent to an appropriate node according to a set of branching probabilities.

Initiation policy determines who invokes load balancing activities under decentralized control. Initiation policy can be sender or receiver initiated. In a sender initiated method, heavily loaded nodes initiate the load transfer. In receiver initiated method, lightly loaded nodes initiate load transfer. The location policy refers to the strategies used for load placement and the transfer policy decides when to transfer the load. Load balancing methods can be adaptive or non-adaptive. Adaptive schemes modify load balancing policies according to system state.

Most literature on dynamic load balancing describes applications in which tasks are dynamically created or arrive at a processor from the external world during run time. However, in the circuit simulation problem the number of tasks is fixed. Load imbalance is caused by differing task sizes during different iterations. Therefore, several techniques described in the literature are not directly applicable to distributed circuit simulation. The load balancing algorithm developed in this thesis is loosely based on *contracting with in neighborhood* and the *gradient model* [75][76]. These approaches are described below; both algorithms described below are distributed in nature. A distributed approach has been selected due to low overhead.

The Contracting Within Neighborhood approach proposed by Kale' [76] defines neighborhoods and horizons of processors in terms of number of hops between processors. All processors exchange load status information with processors within their neighborhood. A newly created task is sent to the least loaded processor within a neighborhood but beyond the immediate vicinity.

Lin and Kellor [75] have proposed a dynamic load balancing scheme based on the *Gradient Model*. In this model, processor load is viewed as a surface. The surface is smoothened by migrating tasks from heavily loaded processors to less loaded processors. A processor can be in one of three states ABUNDANT, IDLE and NEUTRAL depending upon the size of its job queue and memory utilization. Abundant nodes have excess migratable tasks, idle

nodes have few tasks and neutral nodes are neither abundant nor idle. Gradient planes are formed by assuming that idle nodes have the lowest potential. The potential of a node is computed by considering its state and proximity to an idle node. Tasks are initially allocated to the processor on which they are created then load balancing is initiated when a processor becomes idle. The idle processor requests tasks from its neighbors. An Abundant neighbor transfers an excess task to an idle processor. A Neutral node propagates the request for a task to more distant processor. Thus load migration is performed only when necessary. The following section describes a load balancing algorithm used in this thesis.

5.3 Dynamic Load Balancing Algorithm

This section presents a high level overview of the load balancing algorithm used in this research. Important implementation details are presented in the following section. The nature of load imbalance in a distributed queue WR implementation and reasons for selection of the load balancing approach are described below. A distributed queue implementation partitions a sub-circuit task graph into partitions equal to the number of processors. Each processor is assigned a partition for evaluation. During a simulation interval, processors compute a window iteration and synchronize at the end of each window iteration. A processor which completes a window iteration before others remains idle. It waits for all other processors to complete the current iteration.

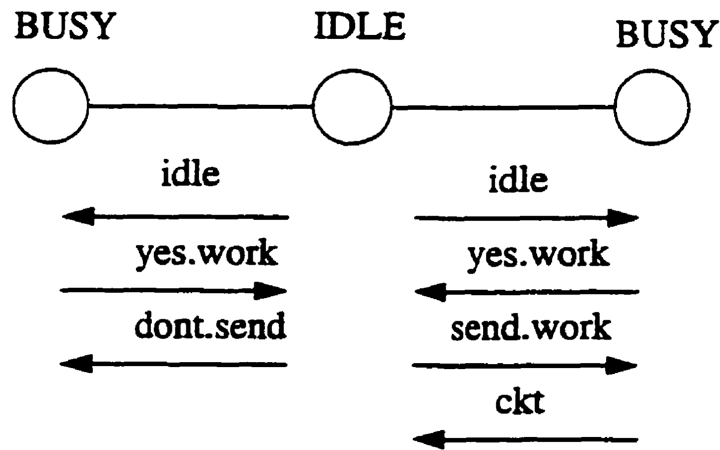
Several load balancing schemes were evaluated for the application to the problem. Load offered by a circuit depends on the nature of the circuit and input conditions. Unlike fluid dynamics problem described by Nichol and Reynolds Jr., no distinctive phases are apparent, therefore periodic re-mapping schemes may not prove effective. Load offered by consecutive waveform iterations of a sub-circuit can be different. Therefore schemes which reassign sub-circuits to processors after a window iteration may not yield good results. The load balancing approach used in this thesis involves temporary migration of sub-circuits from busy processors to idle processors. A high level overview of the load balancing algorithm is given below.

The algorithm is receiver initiated therefore load balancing actions are initiated only when a processor becomes idle. This avoids unnecessary overhead. The algorithm does not depend on the characteristics of Transputers or the Occam language, however the implementation described in the following section considers the synchronous/blocking nature of the inter-process communication. The algorithm takes into account the nature of the application domain. The synchronous/blocking nature of inter-process communication makes implementations prone to deadlocks. Commonly used deadlock resolution schemes involve the application of time-outs. In the circuit simulation problem, *a priori* estimation of sub-circuit evaluation times is difficult, therefore time-outs are not used to avoid deadlocks.

Figure 5.1 shows a pictorial representation of the algorithm. Each processor maintains a list of processors adjacent to it and an idle processor sends an '*I am idle*' message to all its busy neighbors. Busy neighbors return *yes.work* message if they have adequate work to off load. The idle processor sends a *send.work* message to only one busy processor and sends a *dont.send* message to all other busy processors. The busy processor returns a sub-circuit to the idle processor. If a number of *send.work* messages are received then a sub-circuit is sent to only one idle processor and all other processors receive a *no.circuit* message. An idle processor which receives a *no.circuit* message instead of a sub-circuit to evaluate will repeat its request for work. This keeps the algorithm simple. A processor which takes on a sub-circuit for another processor, computes a window iteration for that sub-circuit and returns results to the processor which provided the task. Each round of message exchange is identified by an iteration count in order to avoid ambiguities due to stale messages.

5.4 Implementation of Dynamic Load Balancing

This section describes details of the load balancing algorithm and explains the strategy used for implementation. Procedures relevant to the discussion on load balancing are *sim.remote()* (simulate on a remote node), *check.ckts()* (check circuits) and *sim.rdy.ckts()* (simulate ready circuits). The procedure



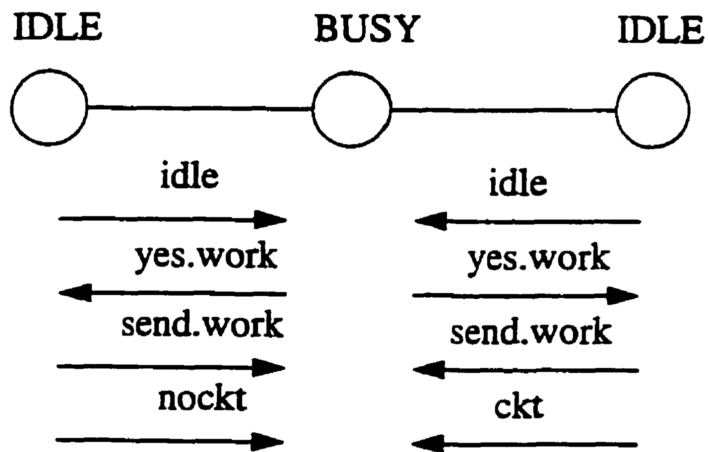


Figure 5.1: A Dynamic load balancing algorithm.

sim.remote() runs on each worker node and calls *check.ckts()* and *sim.rdy.ckts()*. The procedure *check.ckts()* checks sub-circuit dependencies and marks circuits *ready* for simulation. The procedures *sim.rdy.ckts()* simulates ready circuits and implements key parts of the dynamic load balancing algorithm. The dynamic load balancing implementation is explained in terms of these procedures. Dynamic load balancing has been implemented for GJ waveform relaxation, however the framework is generic and is independent of the algorithm used.

Figure 5.2 gives Occam pseudo-code for *sim.remote()*. Several details have been omitted for the sake of clarity. The root transputer sends sub-circuit device lists, sub-circuit initial conditions, sub-circuit-to-processor maps, and sub-circuit fanout lists to each processor. The sub-circuit-to-processor maps are generated by partitioning and allocation modules described in the earlier chapter. The sub-circuit-to-processor map along with processor topology is used to generate distributed routing information.

Sub-circuits are re-ordered in a descending order according to the estimated simulation time. Re-ordering ensures that large circuits are simulated on a local processor. Sub-circuits near the tail of the queue are migrated for load balancing. The WHILE loop used for simulation of sub-circuits consists of PRI ALT and a CASE on tagged messages *start.stop*, *finish*, *values* and a default case TRUE. Details of Occam PRI ALT and CASE constructs are given in Appendix B.

The root transputer sends a *start.stop* message at the beginning of every window along with the start and the end of the window to every worker processor. Upon receiving the *start.stop* message, the worker processor re-orders the sub-circuit queue in a descending order according to the history of execution times. Lists of idle processors and iteration counts are initialized. All sub-circuits ready for simulation are determined and simulated. The *finish* message sent by the root is used to store plot data and terminate the process framework. The *values* message is received from other processors with fan-in sub-circuits. It can satisfy dependencies of some sub-circuits which can be identified and simulated in the default case.

```
sim.remote( ) -- ** Simulate Remote **
 SEQ
   - Receive device list, sub-circuit-to-processor mapping, sub-circuit
     fanout information and details of individual sub-circuits.
   - Rearrange sub-circuits according to size
   WHILE(NOT Done)
     SEQ
       PRI ALT
           input ? CASE
             start.stop; start.time; stoptime
                 SEQ
                     - Rearrange sub-circuits according to history
                     - Initialize idle processor and iteration count arrays.
                     - Check circuits
                     - Simulate ready circuits.
             finish; processor.count
                 SEQ
                     - Send plot data to root. Done set to TRUE.
                     - Shut down process framework.
             values; processor.count
                 SEQ
                     - Receive values from other processors.
                     - Rotate waveforms after a window iteration.
             TRUE & SKIP
                 SEQ
                     - Check circuits.
                     - Simulate ready circuits.
```

Figure 5.2: The sim.remote() (Simulate Remote) procedure.

The *sim.rdy.ckts()* (simulate ready circuits) procedure consists of a SENDER and a RECEIVER WHILE loops. The SENDER loop receives results of an off-loaded sub-circuit, simulates a local sub-circuit and sends status information to adjacent processors. It receives information about the state of adjacent processors and load balancing activities are initiated upon detecting an idle processor. A busy processor (sender) sends a *yes.work* message to idle processors if it has at least two outstanding circuits. It receives *send.work / dont.send* messages from idle processors and sends a circuit to the first processor requesting for work. The SENDER loop attempts to minimize

protocol overhead on a busy processor. The RECEIVER loop is entered after simulating all local sub-circuits. It implements an idle processor protocol described in the previous section. The RECEIVER loop monitors the state of adjacent processors and terminates when all adjacent processors are idle.

```
sim.rdy.ckts(....) -- ** Simulate Ready Circuits **
  SEQ
    WHILE(circuits.to.do > 0)       -- SENDER
      SEQ
        - Receive results from off loaded circuits.
        - Simulate a local sub-circuit.
        - Send status to all neighbors.
        - Receive number and list of idle processors.
        IF
          (number.of.idle > 0)
            SEQ
              - Send yes.work or no.work to adjacent idle processors.
              - Receive send.work or dont.send to from idle processors.
              - Send circuit to the first idle processor requesting work
and
              ...no.circuit to all other idle processors. Do not send
              ...anything to processors that refused work.
              - Send circuit to an idle processor.

    - Receive a number and list of busy processor.
    WHILE(number.busy.processors > 0)        -- RECEIVER
      SEQ
        - Send 'I am idle' message to all busy processors.
        - Accept yes.work or no.work from busy processors.
        - Send send.work to only one procesor and dont.send to all other
          processors.
        - Receive a circuit. Simulate circuit and return results.
        -  Receive a number and list of busy processors.
```

Figure 5.3: The sim.rdy.ckts() (Simulate Ready Circuits) procedure.

5.4 Summary

This chapter describes application of dynamic load balancing techniques to distributed waveform relaxation problems. Limitations of static partitioning techniques are described and the motivation behind using dynamic load balancing techniques is presented. Some existing load balancing techniques are reviewed. A load balancing algorithm is described and important details of the implementation strategy are presented.

6. RESULTS

The primary objectives of this research were to study issues involved in parallel distributed memory Waveform Relaxation based circuit simulation and to analyze the effects of performance limiting factors. Gauss-Seidel and Gauss-Jacobi waveform relaxation programs were developed using a distributed memory parallel programming language, *Occam2*, to study these issues. A Transtech MCP1000 parallel processing board [69], based on the *INMOS IMS T800* Transputer [20], was used to implement the required experiments.

This chapter presents the results obtained from the studies of Gauss-Seidel and Gauss-Jacobi parallel WR programs. A key performance index of interest is the speed-up due to parallel implementation. Results of the sequential version of the program are described below; Section 6.1 gives results for the Gauss-Seidel algorithm, and results for the Gauss-Jacobi algorithm are given in Section 6.2. Section 6.3 presents an analysis of results. In addition, a discussion on the effects of performance limiting factors is presented. Dynamic load balancing was used for circuits which showed low speed-ups due to large load imbalance. Results for load balancing are given in Section 6.4. A summary of this chapter is presented in Section 6.5.

The sequential version of the program was tested using benchmark circuits obtained from the University of California, Berkeley. The following circuits were used.

CINV4, the simplest circuit, consists of a chain of *4* inverters. CINV4 was used to debug the simulator. Relaxation techniques are most effective for simulation of loosely coupled circuits in which node voltages do not depend too strongly on one another. Circuits with tight feedback loops such as, TRINGTX and OPAMP have strongly dependent nodes, therefore application

of waveform relaxation for simulation of these circuits is difficult. The difficulty in simulating these circuits makes them ideal cases to test algorithms.

Table 6.1: Benchmark Circuits.

Name	Devices	Nodes	Sub-circuits
CINV4	10	7	4
DOMINO	8	12	2
TRINGTX	13	6	3
OPAMP	94	13	1

Node voltage waveforms for selected nodes were compared with those obtained using the Relax2.3 program to validate operation of the simulator. Node voltage waveforms for selected nodes obtained using Relax2.3 and the simulator developed in this research are shown in Figure 6.1, Figure 6.2 and Figure 6.3. The mean deviation in results is within typical tolerance limits, 0.01 Volts absolute and 0.01 Volts relative which is adequate for the analysis of MOS digital circuits.

6.1 Gauss-Seidel Method

This section gives results for the parallel Gauss-Seidel waveform relaxation implementation. As described in Section 4.5, the experimental set-up consisted of eight Transputers organized as one root Transputer and seven worker Transputers.

The root Transputer performed initial placement of circuits on worker Transputers and implemented a synchronization barrier at the end of each waveform iteration. Speed-up measurements were done using the measurement instrumentation described in Section 4.5.2.

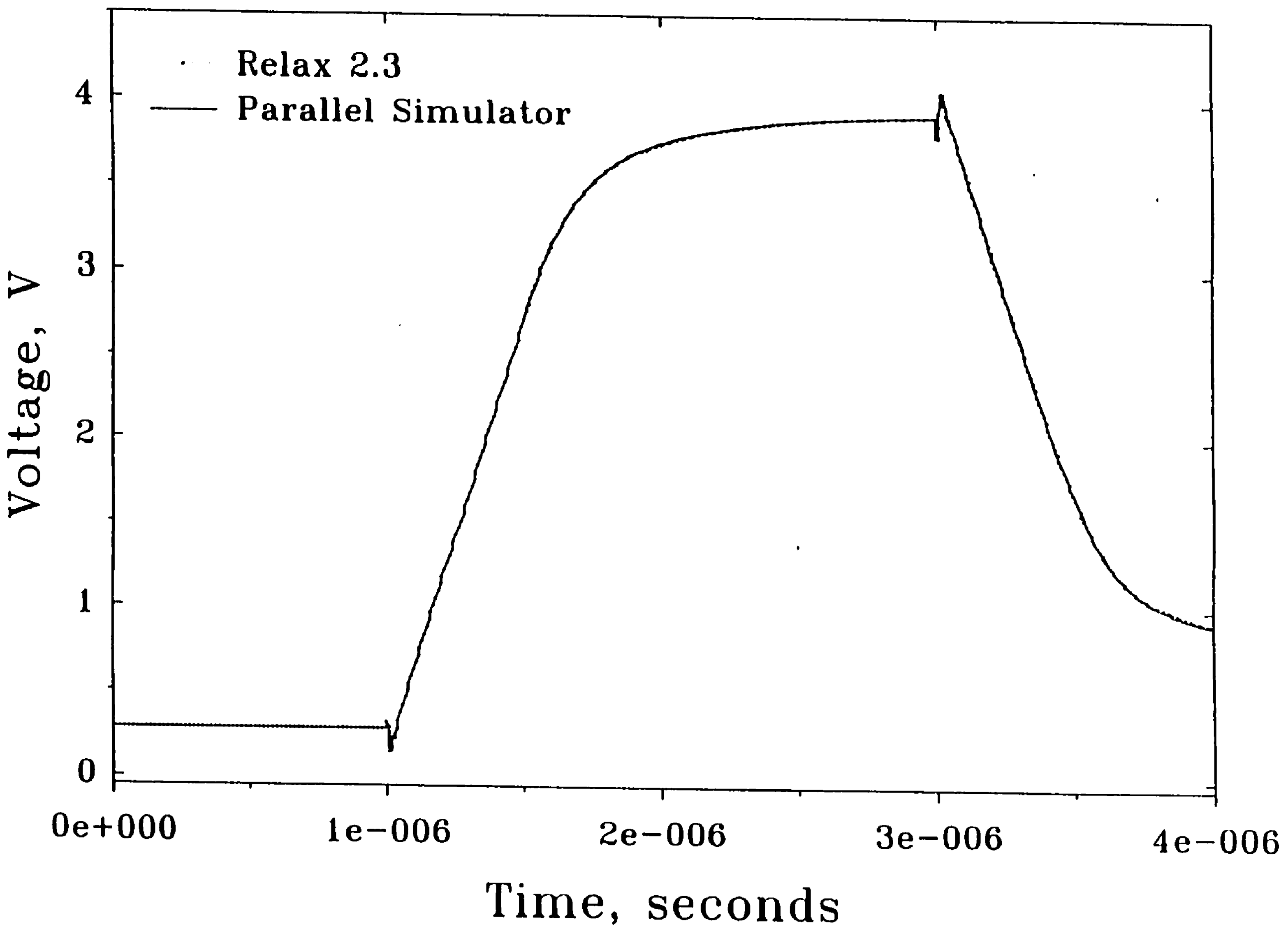


Figure 6.1: Node voltage waveforms for OPAMP.

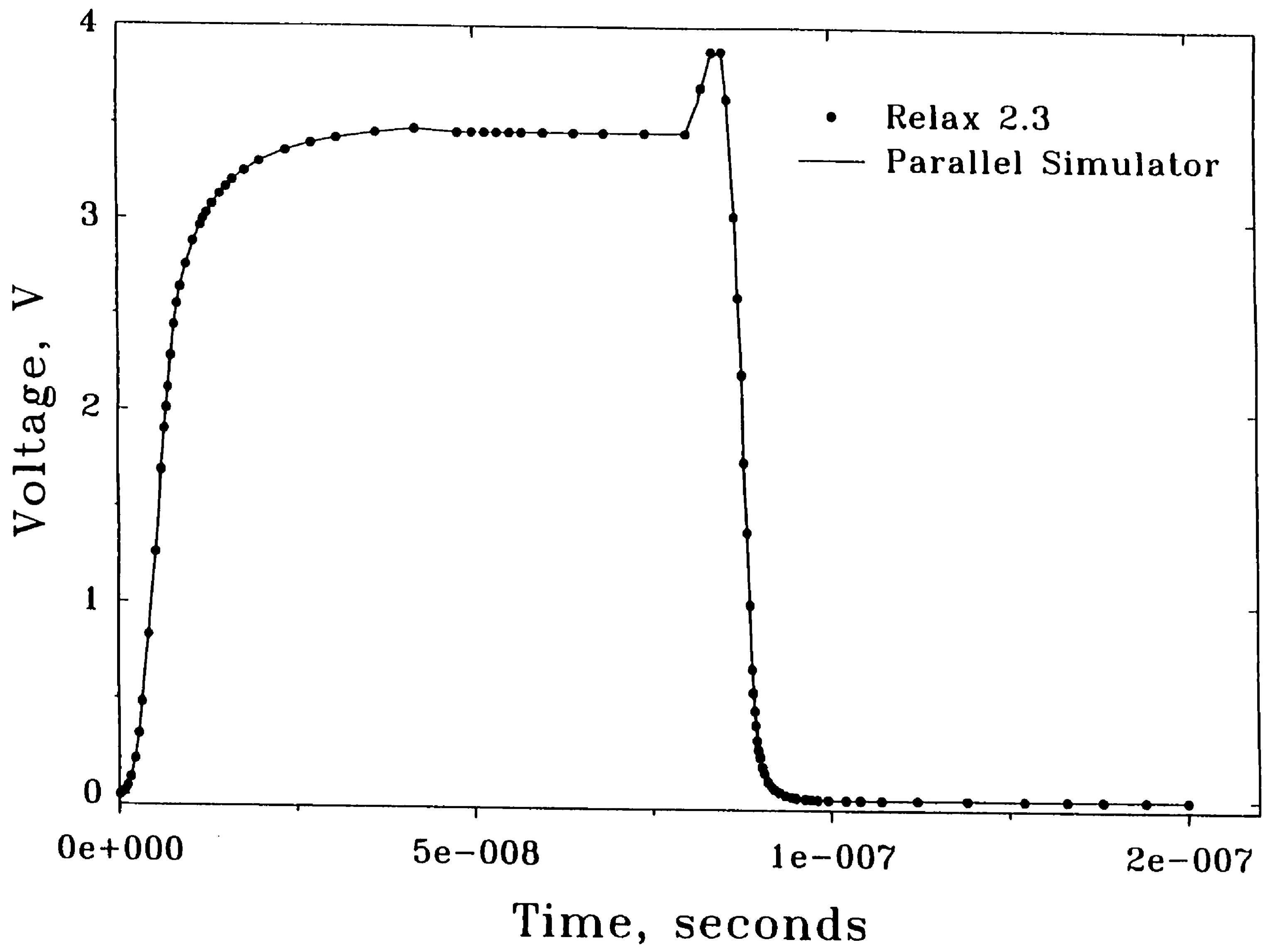


Figure 6.2: Node voltage waveforms for DECPLA.

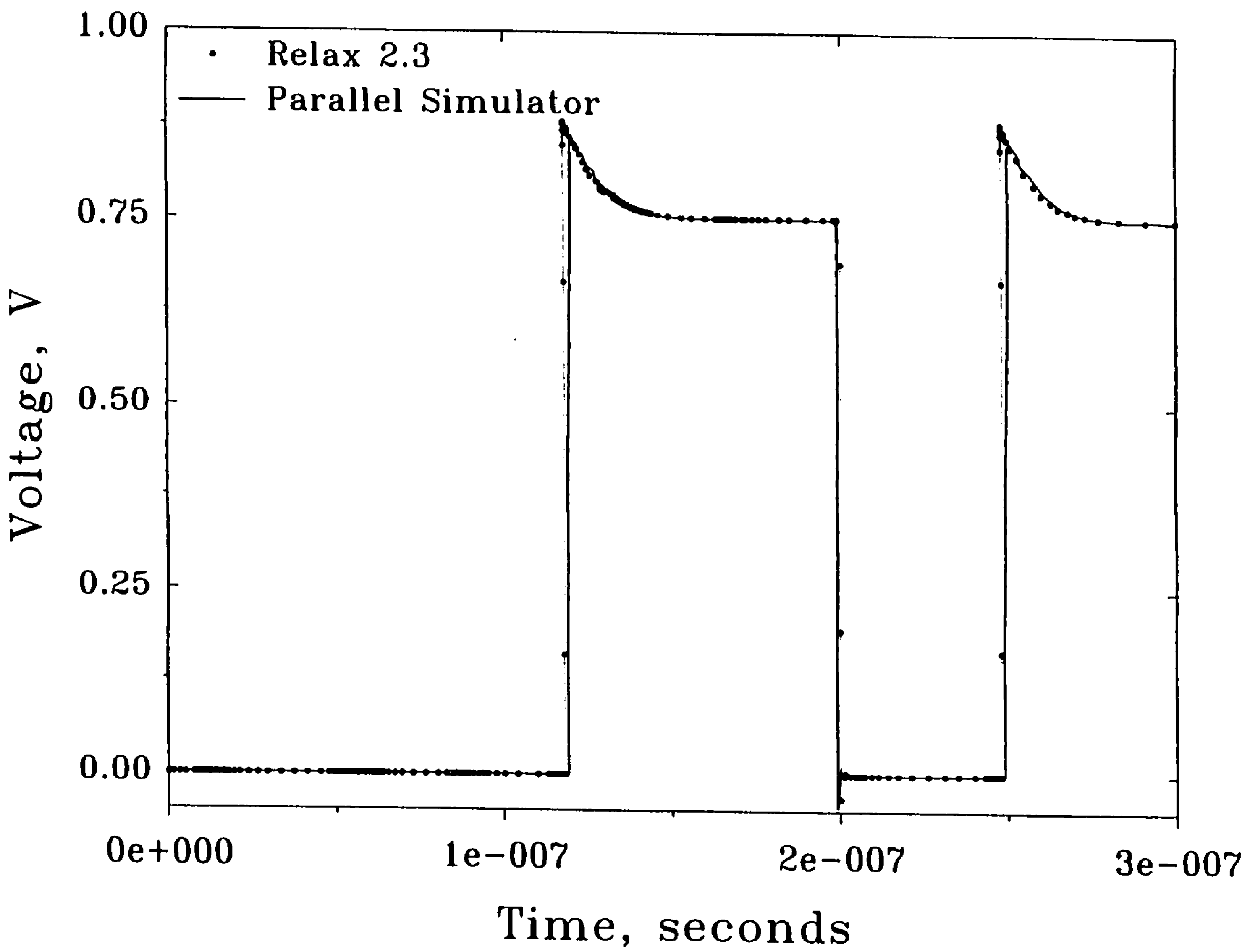


Figure 6.3: Node voltage waveforms for CRAMB.

Speed-up figures for five circuits obtained from the University of Illinois, Urbana Champagne [78] are tabulated below, where speed-up represents the ratio of the time required to simulate a circuit using one processor to the time required to simulate the circuit using seven processors.

Table 6.2: Speed-up for GS algorithm.

Circuit	Nodes	Transistors	Sub-circuits	GS Speed-up
DECPLA	66	116	30	1.54
CRAMB	149	277	76	1.45
DIGFI	385	698	178	2.3
SCDAC	155	416	50	1.84
CKT3	312	428	109	3.02

Parallelism in GS waveform relaxation is limited due to dependencies among sub-circuits of a given iteration. However the Gauss-Seidel algorithm is suited for a class of circuits which have strong directional properties. Speed-up for the first two circuits, DECPLA and CRAMB is low. The DECPLA circuit has very few sub-circuits. Therefore starvation and load imbalance cause low speed-up. CRAMB has one very large sub-circuit and uneven sub-circuit sizes result in reduced in parallelism and therefore low speed-up.

6.2 Gauss-Jacobi Method

Analysis of parallel versions of Gauss-Seidel and Gauss-Jacobi waveform relaxation algorithms shows that parallel Gauss-Jacobi is asymptotically faster than parallel Gauss-Seidel when a sufficiently large number of processors are used [79]. Distributed memory machines are usually built using a large number of relatively less expensive processors. Therefore GJ is an important algorithm. Results for the Gauss-Jacobi case are tabulated below.

Reasons for lower speed-up for DECPLA and CRAMB are similar to the GS case. Load imbalance is a potential source of overhead. A detailed analysis of overheads is presented in the following section.

6.3 Analysis of Performance Limiting Factors

This section presents a simple model to analyze the effects of performance limiting factors. Three performance limiting factors: *communication*, *sequential fraction* and *load imbalance* are analyzed. Communication is an important source overhead in distributed memory machines, therefore a detailed discussion of communication overhead is presented below. Inter-processor communication can be divided into two categories: communication among processors to exchange node voltage waveforms and communication among root and worker Transputers to determine the size of the next window. Both categories are analyzed. *INMOS Transputer T800* communication mechanisms and their costs obtained from [73] and [80] are used to illustrate concepts.

Table 6.3: Speed-up for GJ algorithm.

Circuit	Nodes	Transistors	Sub-circuits	GJ Speed-up
DECPLA	66	116	30	3.91
CRAMB	149	277	76	2.64
DIGFI	385	698	178	4.67
SCDAC	155	416	50	4.33
CKT3	312	428	109	5.53

6.3.1 Communication Overhead

The cost of communication between adjacent Transputers can be represented as the sum of a fixed cost α and an incremental cost per unit length β. The fixed cost is the start-up time for any message. The initial part of analysis assumes that the cost of sending a message is equal to the start-up cost and neglects the incremental cost.

The Transputer links are autonomous DMA engines. Links permit bi-directional transfer of data without seriously degrading the performance of the processor [73]. The run-time environment described in Chapter 4 de-couples computation assigned to a processor from its link communication. Communication processes are run at a higher priority and communication is overlapped with computation. Under these circumstances, the incremental cost is incurred mainly due to DMA cycle stealing. In this work, however, the incremental cost due to cycle stealing has been neglected to simplify the analysis without compromising accuracy. The worst case analysis of the DMA cycle stealing overhead is given at the end of the section.

The notation, symbols and assumptions used for the analysis are described below. The number of processors in the system is denoted by p and the number of circuits is denoted by c. A uniform allocation of circuits to processors is assumed to simplify analysis so the number of circuits per processor is $l = c/p$. Each circuit communicates with f_c other sub-circuits residing on other processors. This is known as a sub-circuit fanout. Therefore fanout per processor f_p is given by $f_p = f_c * l$ and each processor sends f_p messages during each window iteration. An average distance traveled by a message is denoted by a_d hops. The analysis does not take into account the effects of locality in message passing.

Analysis of inter-processor communication to exchange node voltage waveforms is presented below. The majority of inter-processor messages are to exchange node voltage waveforms, therefore this form of communication is the most important source of communication overhead. The total message establishment cost for a window iteration is given by:

$$\textit{Message Establishment Cost} = 2 * f_p * p * a_d \quad \textit{message establishments.}$$

Both processors involved in the communication incur message establishment cost, therefore the multiplication factor *2* appears in the equation. Message establishment cost per processor is given by:

$$\textit{Cost per Processor} = 2 * f_p * a_d \ \textit{message establishments.}$$

The analysis given below can be divided into two parts: overhead analysis for the eight Transputer system (one root and seven workers) and estimation of the effect of increasing the number of processors. Overhead analysis for the eight transputer system is given below. A synthetic circuit used for analysis has *70* sub-circuits and the complexity of each sub-circuit is approximately equal to an average DECPLA sub-circuit. Each sub-circuit communicates with 3 other sub-circuits which reside on other processors. A partitioning algorithm partitions the circuit into 7 equal partitions, therefore each processor is assigned *10* sub-circuits to evaluate. Measurements done on DECPLA indicate that the average window evaluation time for a sub-circuit using the *INMOS T800* processor is *9600* micro-seconds. Therefore *10* sub-circuits require *96000* micro-seconds to complete a window iteration. The average distance traveled by a message for the 2×4 mesh topology (see Figure 4.1) is *2* hops.

The per processor of message establishment cost is therefore given by:

$$Cost\ per\ Processor = 2 * f_p * a_d = 2*(3*10)*2 = 120\ message\ establishments.$$

Boreddy and Pulraj have measured message establishment costs for *INMOS T800* Transpurers under different operating conditions [80]. Their results indicate that the average message establishment cost is *25* micro-seconds. Therefore the message establishment cost for a window iteration for a processor is 3000 micro-seconds. This is less than 3.5% of the computation cost of a window iteration on a processor.

The analysis of communication overheads presented above can be extended to study the effect of increasing the number of processors. It is assumed that the problem size grows with the number of processors so the number of sub-circuits assigned to a processor remains constant. A *2* dimensional torus network with even width is used for the analysis. The average message path length of a w-wide, D-dimensional torus is:

$$a_d = \frac{w * D}{4} \quad for\ w\ even\ [39].$$

The following table shows the variation of the message passing cost with an increase in the number of processors.

The communication cost increases linearly with the number of processors, therefore appropriate partitioning and allocation to exploit locality of message communication is necessary. In addition, if the problem size does not scale with an increase in the number of processors then the communication cost as a percentage of computation cost will increase.

Table 6.4: Variation of Communication Cost.

p	$a_d = \frac{w}{2}$	Cost per processor in micro-seconds	Communication cost as a percentage of the computation cost
16	2.0	3000	3.12
36	3.0	4500	4.68
64	4.0	6000	6.25

The analysis presented above assumes that the cost of sending a message is equal to the start-up cost and ignores the incremental cost. The incremental cost, in this case, is mainly due to DMA cycle stealing overhead. Once transfer over a link is started it typically consumes 4 processor cycles (0.2 micro-seconds), to perform one memory read or write per 32-bit word every 4 micro-seconds [73]. Therefore each link consumes *5* percent of the memory time. This cycle stealing can cause the CPU to stall until the transaction is complete.

Another form of communication involves the communication between the root and the worker processors since the root communicates information about the next window to worker processors. This communication is not overlapped with computation, however, the message size is usually very small (typically 80 bytes or less). In addition, a form of broadcast communication is used. The root communicates values to adjacent processors which propagate the values to their neighbors. An estimate of the worst case latency to the farthest processor is given by:

$$L = (\alpha + \beta * M) * \textit{Length of the longest path},$$

where M is the length of the message in bytes, α is the start-up cost in micro-seconds, β is the incremental cost in bytes per micro-second and the length of the longest path is expressed in terms of the number of hops. The length of the longest path for the 2×4 mesh network is *4*. Assuming $\alpha = 25$ micro-seconds, $\beta = 0.80$ micro-seconds per byte [80], and $M = 80$ bytes gives:

$$L = (25 + 64) * 4 = 356 \text{ micro-seconds}.$$

The estimated worst case latency is approximately *0.37* percent of the time required for computing a window iteration, which is very low.

6.3.2 Sequential Computation

The sequential computation performed by the root Transputer involves computing the size of the next window. The root Transputer accepts information such as the maximum number of time points required to describe waveforms and convergence status from worker processors. The algorithm used for computing the size of the next window is presented in Section 2.5.1. Timing measurements done for the DECPLA circuit indicate that 128 micro-seconds are required to determine the size of the window which is *0.13* percent of the time required for a window iteration. The sequential computation required to calculate the size of the next window does not depend on the number of sub-circuits, however the root Transputer receives a message from every processor and two comparisons per processor are performed. Messaging from worker Transputers to the root could be easily reduced by combining values sent from worker processors to the root at intermediate processors. Therefore sequential fraction is not a major source of overhead.

6.3.3 Load Imbalance Overhead

The time required to simulate a sub-circuit depends on the nature of the sub-circuit and the input excitation conditions. Also sub-circuit execution times vary widely over iterations. These factors make *a priori* estimation of sub-

circuit sizes and accurate static partitioning difficult. Therefore some processors complete their window iteration before others, and remain idle. Load imbalance overhead depends highly on the nature of the circuit, for example, DECPLA and CRAMB show relatively low speed-ups.

Run-time traces for DECPLA and CRAMB were collected to determine the reasons for low speed-ups. The run-time trace for a processor consisted of the following information: window iteration number, time required to complete the window iteration, and time required to simulate its sub-circuits during the window iteration.

Run-time traces collected from worker processors were stored at the root processor. Post-processing of the run-time traces was done by an analysis program, which generated statistical information on that record. The program calculated aggregate processor idle time for a window iteration using the following equation:

*Aggregate processor idle time = Number of proccessors * Time spent by the slowest processor - Sum of individual sub-circuit execution times.*

The total idle-time for the entire simulation interval was calculated by summing-up the aggregate processor idle time for all the windows. The aggregrate processor idle time for DECPLA and CRAMB is given in Table 6.4.

Table 6.4: Aggregrate Processors Idle Time

Circuit	Algorithm	Aggregrate processors idle time (%)
DECPLA	GS	75.80
DECPLA	GJ	35.01
CRAMB	GS	78.53
CRAMB	GJ	55.76

The result of the experiment presented in Table 6.4 clearly demonstrates that the performance of the multi-computer system is severely affected by idling of processors. This arises due to load imbalance and starvation conditions.

6.4 Dynamic Load Balancing

The dynamic load balancing algorithm described in Chapter 5 relies on temporary migration of sub-circuits from busy to idle processors to reduce the effects of load imbalance. The performance of this algorithm is determined by implementing it on a network of Transputers. The performance of the algorithm is measured by simulating test circuits.

The parallel GJ waveform relaxation simulator with a dynamic load balancing module was used to simulate the DECPLA and CRAMB circuits. As discussed in the previous section, these circuits have a large load imbalance overhead, which makes them appropriate cases to test the load balancing algorithm.

The procedure used to verify the load balancing algorithm can be divided in three steps. The first step is to identify the window iterations in which sub-circuit migrations take place and processors which off-load and which receive sub-circuits. The second step obtains run time traces for the selected window iterations and appropriate processors. The third step verifies if migrations occur according to the algorithm described in Chapter 5.

The information used to identify iterations with migrations included run time traces for the parallel GJ algorithm without the load balancing module and the processor adjacency list. The run time traces for the parallel GJ algorithm described in Section 6.3.3 provided the following information: window iteration number, time required to complete the window iteration, and time required to simulate the individual sub-circuits during the window iteration. The adjacency list of a processor provides node numbers of all neighbors. Migrations take place when a busy processor has more than one outstanding sub-circuit and it detects an idle neighbor. Run-time traces for selected iterations and appropriate processors were obtained. They show

migrations of sub-circuits in accordance with the algorithm described in Chapter 5.

Five test runs for DECPLA and CRAMB each were performed in order to account for small variations among test runs. All the test runs clearly indicated that there was no appreciable gain in speed-up. The best possible speed-up for DECPLA was 3.93, which was not significantly higher than the 3.91 that was achieved without dynamic load balancing.

Reasons for low gain due to load balancing are described below. The dynamic load balancing algorithm permits migration of circuits only when the number of outstanding circuits on a processor is more than one. Each partition of a DECPLA consists of only 4 or 5 sub-circuits, so very few circuit migrations can occur. Circuits are already arranged in a decreasing order of execution times, so only small circuits migrate. In the case of CRAMB, load imbalance is mainly due to one large sub-circuit. It is important to note that the gain due to application of dynamic load balancing depends on the nature of the circuit and number of processors.

6.5 Summary

This chapter presents speed-up results for parallel GS and parallel GJ waveform relaxation programs. Three performance limiting factors: communication, sequential fraction, and load imbalance are analyzed. The communication overhead depends on the architecture of the multi-computer system and the nature of the circuit. Transputer links are autonomous DMA engines and the effect of communication overhead can be minimized by effectively overlapping computation and communication. Analysis and measurement of the sequential part of the program shows that it is not a major source of overhead for a full window WR technique. Load imbalance is the largest source of overhead. Results for dynamic load balancing techniques are presented.

7. SUMMARY, CONCLUSIONS AND FUTURE WORK

This chapter presents a summary of the research described in this thesis. Objectives of the research are reviewed and main contributions are summarized. Finally possible future work is discussed.

7.1 Summary

This thesis describes analysis, implementation and performance evaluation of a distributed memory parallel waveform relaxation technique for the simulation of MOS VLSI circuits.

Detailed electrical circuit simulation is commonly used to simulate MOS VLSI circuits to verify functionality and to predict their performance before fabrication. Electrical circuit simulators perform ac, dc and time domain transient analysis of circuits. The time domain transient analysis provides accurate timing information and performance details, however the time required for the simulation of a circuit consisting of a few thousand transistors can be several CPU hours. This research was, therefore, aimed at finding the techniques to improve the speed of electrical circuit simulators, without sacrificing the accuracy of analysis, so that a substantial reduction in development cycle time and cost of integrated circuits could be achieved.

Transient analysis requires solution of a system of nonlinear algebraic-differential equations. Equations are formulated using Kirchoff's current law, Kirchoff's voltage law and branch relations, and solved using direct or iterative techniques. Waveform relaxation is an iterative technique for the solution of a system of nonlinear algebraic-differential equations. This technique transforms a system consisting of n coupled differential equations into n equations consisting of one variable each. Each de-coupled equation is independently solved and Gauss-Jacobi or Gauss-Seidel techniques are used to iterate the equations.

Waveform relaxation based circuit simulators provide waveforms as accurate as those of a standard circuit simulator, with up to two orders of magnitude speed improvement for large circuits. The speed improvement is obtained by exploiting waveform latency and multi-rate behavior.

Prior research on the waveform relaxation technique is reviewed in Chapter 2. It presents circuit partitioning and window selection techniques, which enhance the robustness and speed of the basic waveform relaxation technique. Various partitioning schemes such as, functional extraction, dc component (dcC) partitioning and Norton equivalent conductance partitioning are described. Isolation of tightly coupled nodes into sub-circuits aids convergence, therefore Norton equivalent conductance partitioning is used in this research. The window selection technique divides the simulation interval into a number of sub-intervals known as windows. Each of these windows is simulated using the waveform relaxation technique. This scheme is particularly useful for simulation of circuits with logic feedback loops. A heuristic technique to determine the size of a window is presented. The direct method used in this research for simulating sub-circuits is also described in Chapter 2.

The waveform relaxation technique exhibits inherent parallelism due to the partitioning of a circuit into a number of sub-circuits. These sub-circuits can be concurrently simulated on parallel processors. In addition, the full window waveform relaxation technique permits exchange of large and infrequent message among sub-circuits. This feature is useful for parallel implementation on distributed memory machines.

The issues and options involved in the parallel implementation of relaxation based circuit simulators are described in Chapter 3. Important classes of parallel architectures such as shared memory and distributed memory are described and design methodologies for parallel applications are studied. The shared memory programming model is more similar to the uniprocessor programming model, therefore these machines are easier to program. In addition, their lower inter-processor communication cost allows exploitation of fine grain parallelism. Shared memory architectures are, however, complex and expensive, and synchronization for shared resources limits their scalability. The distributed memory programming model is substantially

different from the shared memory programming model. Distributed memory computers are relatively less expensive and scale well, however it is usually difficult to program distributed memory computers. Partitioning, allocation, and load balancing are usually done by the application programmer.

The performance of parallel applications is limited due to *sequential fraction*, *communication overhead,* and *load imbalance.* The fraction of the computation that must be performed sequentially limits the number of processors that can be usefully put to work on a given problem. Communication and load imbalance can be represented as additional work. It is observed that an algorithm that causes some extra work can still be useful for implementation on distributed memory computers if it has a low sequential fraction. This is mainly due to the low cost of adding extra processors. A fairly low speed-up per processor can be tolerated as long as the execution time decreases with an increase in the number of processors.

The rest of Chapter 3 is devoted to the analysis of parallelism in circuit simulation. Coarse and fine grain parallelism in the direct and relaxation methods are analyzed. Waveform relaxation techniques have two forms of parallelism: coarse grain parallelism across sub-circuits and fine grain parallelism within a single Newton-Raphson iteration of a sub-circuit. Techniques used to exploit both forms of parallelism are described. Tradeoffs between parallelism and complexity thread of control are studied. The discussion concentrates mainly on parallel implementation of waveform relaxation programs on shared memory multi-processors. Partitioning and allocation techniques specific to distributed memory machines are also described.

Chapter 4 presents an implementation of a parallel waveform relaxation program. Different strategies for parallel implementation such as a single queue approach and a distributed queue approach are described and compared. The distributed queue approach is selected for exploiting the coarse grain parallelism across sub-circuits. This approach is more generic and scales well.

A distributed queue implementation involves static partitioning and placement of sub-circuits on processors. Static partitioning techniques use

task execution and inter-task communication times to partition a task graph. Very little *a priori* information about task execution and inter-task communication times is available, therefore the program uses an algorithm based on the critical path method to partition a GS task graph. This algorithm attempts to generate partitions with a high degree of parallelism. An algorithm based on bin packing heuristics is used to partition a GJ task graph. The objective of these algorithms is to reduce the effect of load imbalance overhead. The two algorithms discussed above neglect the effects of communication overhead, therefore a placement heuristic which attempts to minimize the communication overhead is used.

The parallel processing framework provides a skeleton for implementing parallel applications. An efficient implementation of the processing framework which exploits characteristics of the underlying hardware is necessary to improve CPU utilization. Therefore, the parallel processing framework is discussed in detail.

An imbalanced workload in a parallel processing system results in low overall efficiency and speed-up. The load offered by a circuit simulator when simulating a large digital circuit changes with simulation time due to latency and multi-rate behavior. Therefore, load imbalance is an important source of overhead. Chapter 5 presents dynamic load balancing techniques used to reduce the load imbalance;. several load balancing schemes were evaluated for application to the problem. The load balancing approach developed in this thesis involves temporary migration of sub-circuits from a busy processor to its idle neighbors. The algorithm is receiver initiated. Therefore load balancing actions are initiated only when a processor becomes idle. This avoids unnecessary overhead. An implementation using a multi-transputer system is also presented.

Results of implementations described in Chapters 4 and 5 are given in Chapter 6. Six benchmark circuits and seven worker processors were used to measure speed-up results for parallel GS and parallel GJ waveform relaxation programs. The speed-up for parallel GS varied from 1.45 to 3.02 and the speed-up for parallel GJ varied from 2.64 to 5.53. Three performance limiting factors: communication, sequential fraction, and load imbalance are analyzed. The communication overhead depends on the architecture of the

multi-computer system and the nature of the circuit. Transputer links are autonomous DMA engines and the effect of communication overhead can be minimized by effectively overlapping computation and communication. Analysis and measurement of the sequential part of the program shows that it is not a major source of overhead for a full window WR technique. Load imbalance is the largest source of overhead. Results for dynamic load balancing techniques are presented, however no appreciable gain in speed-up was observed for the cases studied. Conclusions drawn from the research are presented in the following section.

7.2 Conclusions

The principal objective of this research was to study issues involved in the application of distributed memory parallel processing for the simulation of VLSI circuits using waveform relaxation techniques. A specific aim of the thesis was to determine how much the speed of calculation in circuit simulation can be increased with a low cost distributed memory parallel processing system. Also, a method to efficiently implement circuit simulation on the distributed memory machine was sought.

Different forms of parallelism in the direct method and the waveform relaxation technique were studied. An analysis of a single queue and distributed queue approaches to implement parallel waveform relaxation on distributed memory machines was performed and their performance implications were studied. The distributed queue approach was selected for exploiting the coarse grain parallelism across sub-circuits. A distributed queue implementation involves static partitioning and placement of sub-circuits on processors. An algorithm based on the critical path method and an algorithm based on bin packing heuristics were used to partition GS and GJ task graphs respectively. Parallel waveform relaxation programs based on Gauss-Seidel and Gauss-Jacobi techniques were implemented using a network of one root and seven worker Transputers. Static and dynamic load balancing strategies were studied. A dynamic load balancing algorithm was developed and implemented. Results of parallel implementation were analyzed to identify sources of bottlenecks.

This thesis has demonstrated the applicability of a low cost distributed memory multi-computer system for simulation of MOS VLSI circuits. Speed-up measurements prove that a five times improvement in the speed of calculations can be achieved using a full window parallel Gauss-Jacobi waveform relaxation algorithm. Analysis of overheads shows that load imbalance is the major source of overhead and the fraction of the computation which must be performed sequentially is very low. Communication overhead depends on the nature of the parallel architecture and the design of communication mechanisms. The run-time environment (parallel processing framework) developed in this research exploits features of the transputer architecture to reduce the effect of the communication overhead by effectively overlapping computation with communications, and running communication processes at a higher priority.

The main contributions made by this thesis are:

1. the development of the first method to implement waveform relaxation on a low cost distributed memory machine,
2. the analysis of overheads and performance limiting factors, and
3. the development and implementation of a dynamic load balancing algorithm.

This research will contribute to the development of low cost, high performance workstations for computer-aided design and analysis of VLSI circuits.

7.3 Future Work

Several interesting studies could be usefully conducted to further improve and extend the results of this research.

Gains due to dynamic load balancing depend on the nature and size of the circuit and the available number of processors. Most benchmark circuits used in this research were small. Dynamic load balancing should be tested on larger circuits and using more processors. The multi-computer system used in this research had only *4* Megabytes of memory on each node which made simulation of large circuits difficult.

Granularity of window iterations in a full window waveform relaxation technique is coarse. This increases the effect of load imbalance. Schemes which dynamically change the granularity depending on the circuit size and the number of available processors seems promising. Fine grained computations due to parallel model evaluation and time segment pipeline techniques could be used in conjunction with coarse grained parallelism to balance load more effectively. In addition, dynamic processor partitioning techniques used in the multi-programmed multi-computer environment could be employed to improve utilization of processors.

Parallel ITA could be implemented on distributed memory machines. It uses event driven selective trace techniques to schedule sub-circuits. This would make static partitioning and dynamic load balancing challenging.

Shared and distributed memory machines represent two extremes styles of parallel architectures. Several parallel architectures have been proposed which share attributes of shared and distributed memory machines. Issues involved in parallel implementation of circuit simulation programs on these architectures should be studied.

REFERENCES

1. Lee James M., *Verilog Quick Start*, Kulwer Academic Publishers, Boston 1997.

2. Mukherjee A., *Introduction to nMOS and CMOS VLSI Systems Design,* Prentice-Hall, Englewood Cliffs, NJ 07632.

3. Newton A.R., Sangiovanni-Vincenttelli Alberto L., "Relaxation-Based Electrical Simulation," *IEEE Transactions on Computer-Aided Design*, Vol. CAD-3, October 7 1984, pp. 308-331.

4. Nagel W., SPICE2: *A Computer Program to Simulate Semiconductor Circuits*, Electronics Research Lab. Report No. ERL-M520, University of California, Berkeley, University of California Berkeley, 1975.

5. Weeks W. T., Jimenez A. J., Mahoney G. W., Metha D., "Algorithms for ASTP- A Network Analysis Program," *IEEE Transactions on Circuit Theory*, Vol. CT-20, November 1973, pp. 628-634.

6. Yang P., Hajji I. N., and Trick T.N., "SLATE: A Circuit Simulation With Latency Exploitation and Node Tearing," *Proceedings of the IEEE International Conference on Circuits and Computers*, 1980.

7. Quarles T. L., *Analysis of Performance and Convergence Issues for Circuit Simulation*, Ph.D.. dissertation, Dept. of Electrical Engineering and Computer Science, University of California Berkeley, 1989.

8. Saleh R. A. and Newton A. R., *Mixed-Mode Simulation*, Kulwer Academic Publisher, Boston, USA, 1990.

9. Lelarasmee E., *The Waveform Relaxation Method for Time Domain Analysis of Large Scale Integrated Circuits*, Ph.D. dissertation, University of California, Berkeley, 1982.

10. White J., *Multirate Integration Properties of Waveform Relaxation With Applications to Circuit Simulation and Parallel Computation*, Ph.D. dissertation, University of California, Berkeley 1986.

11. Saleh R. A., *Nonlinear Relaxation Algorithms for Circuit Simulation*, Ph.D. dissertation, University of California, Berkeley, 1986.

12. Matisson S., *CONCISE: A Concurrent Circuit Simulation Program*, Ph.D. dissertation, Dept. of Applied Electronics, University of Lund, Sweeden, 1986.

13. Deutch J. T., *Algorithms and Architectures for Multiprocessor-Based Circuit Simulation*, Ph.D. dissertation, Uniersity of California, Berkeley, 1985.

14. White Jacob K., Alberto Sangovanni-Vincentelli, *Relaxation Techniques for Simulation of VLSI Circuits*, Kulwer Academic Publishers, Boston, USA, 1987.

15. Stone Herald, *High-Performance Computer Architecture*, Addison-Wesley Pub. Co., Reading Mass., 1990.

16. Duncan R. "A Survey of Parallel Computer Architectures," *IEEE Computer*, Vol 23, No 2, February 1990, pp. 5-17.

17. Flynn M. J., "Some Computer Organizations and Their Effectiveness," *IEEE Transations on Computers*, Vol. C-21, September 1972, pp. 948-960.

18. Hwang K. and Briggs F. A., *Computer Architecture and Parallel Processng*, McGraw Hill, New York, 1984.

19. Seitz C. L., "The Cosmic Cube," *Communications of the ACM,* January 1985, Vol 28, No. 1, pp. 22-23.

20. The Transputer Databook, INMOS Limited, Prentice-Hall 1988..

21. Burns J. L., Newton A. R., Pederson D. O., "Active Device Table Look-up Models for Circuit Simulation," *Proceedings of the 1983 Int. Symp. on Circuits and Systems, 1983.*

22. Chawla B. R., Gummel H. K., Kozak P., "MOTIS: a MOS Timing Simulator," *IEEE Transactions on Circuits and Systems*, Vol. CAS-22, Dec. 1975, pp. 901-909.

23. Cohen E., *Performance Limits of Integrated Circuit Simulation on a Dedicated Minicomputer System*, UCB/ERL M81/29, University of California Berkeley, Electronics Research lab., University of California Berkeley, 1981.

24. Sangiovanni-Vincentelli, Chen L.K., Chuo L. O., "A New Tearing Approach - Node Tearing Nodal Analysis," *IEEE International Symp. on Circuits and Systems,* Vol. I, 1977, pp. 143-147.

25. Valdiminrescu, *LSI Circuit Simulation on Vector Computers,* Ph.D. dissertation, University of California, Berkeley, 1982.

26. McCalla William J., *Fundamentals of Computer-aided Circuit Simulation,* Kulwer Academic Publishers, Boston, 1988.

27. Saleh R. A., Gallivan K. A., Mi-Chang Chang, Hajji I. N., Smart David, and Trick T. N., "Parallel Circuit Simulation on Supercomputers," *Proceedings of the IEEE*, Vol. 77, No. 12, December 1989, pp.1915-1931.

28. Nagel L. W. and Rohrer R. A., "Computer Analysis of Nonlinear Circuits, Excluding Radiation (CANCER)," *IEEE Journal of Solid-State Circuits*, Vol. SC-6, August 1971, pp 166-182.

29. Chua L., Lin P., *Computer-Aided Analysis of Electronic Circuits: Algorithms and Computational Techniques*, Prenice-Hall, 1975.

30. Valch J. and Singhal K., *Computer Methods for Circuit Analysis and Design*, CBS Publisher and Distributors, Delhi-110032 (INDIA).

31. Ho C. W., Ruheli A. E., Brenan P. A., "The Modified Nodal Approach to Network Analysis," *IEEE Transactions on Circuits and Systems*, Vol. CAS-22, June 1975, pp. 504-509.

32. Sangiovanni-Vincentelli A., *Circuit Simulation in - Computer Design Aids for VLSI Circuits*, P. Antognetti, Peterson D. O. and De Man H., Groningen, The Netherlands: Sijthoff and Noordhoff, Netherlands, 1981, pp. 19-113.

33. Brayton R. L., Gustavson F. G., Hatchtel G. D., "A New Efficient Algorithm for Solving Differential-Algebraic Systems Using Implicit Backward-Differentiation Formulas," *Proceeding of the IEEE*, Vol. 60, No. 1, Jan. 1972, pp. 98-108.

34. Verga J., *Matrix Iterative Analysis*, Prentice-Hall, Englewood cliffs, NJ:, 1962.

35. Ortega J. M. and Rheinbolt W. C., *Iterative Solutions of Nonlinear Equations in Several Variables*, New York: Academic Press, New York, 1970.

36. Marong G. and Sangiovanni-Vincentelli, "Waveform Relaxation and Dynamic Partitioning for Transient Simulation of Large Scale Bipolar Circuits," *International Conference in Computer-Aided Design*, Santa Clara, CA, Nov. 1985.

37. Debefve Paul, Odeh F. and Ruehli, "Waveform Techniques" in *Circuit Analysis, Simulation and Design, 2*, Elsevier Science Publishers B. V. North-Holland, 1987.

38. Carlin C. H., Vachoux A., "On Partitioning for Waveform Relaxation Time-Domain Analysis of VLSI Circuits," *International Symposium on Circuits and Systems*, Montreal, Canada, May 1984.

39. Zang X., Castaneda R. and Chan E. W., "Spin-lock Synchronization on Butterfly and KSR1," *IEEE Parallel and Distributed Technology*, Spring 94.

40. Gordon Bell, "Ultracomputers: A Tetraflop Before its Time," *Communications of the ACM*, Vol. 35, No. 8, August 1992, pp. 26-47.

41. Hwang Kai, *Advanced Computer Architecture: Parallelism, Scalability, Programmability*, McGraw-Hill series in Computer Science, McGraw-Hill Inc. 1993.

42. El-Rewini H., Lewis T. G. and Ali H. H., *Task Scheduling in Parallel and Distributed Systems*, Prentice-Hall 1994.

43. Amdahl G., "The Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities," *AFIPS Conference Proceedings*, Vol. 30, 1967.

44. P. Sadayappan, V. Visvanathan, "Circuit Simulation on a Multiprocessor," *Proceeding of the Custom Integrated Circuit Conference*, Portland, OR, May 1987, pp. 124-128.

45. O. Wing and J.W. Huang, "A Computation Model for Parallel Solution of Linear Equations," *IEEE Transactions on Computers*, Vol. C-29, 1980, pp. 632-638.

46. George A. and Liu J. W. *Computer Solution of Large Positive Definite Systems*, Englewood Cliffs, NJ: Prentice Hall, 1981.

47. Noor A., Kamal H.and Fulton R. "Substructuring Techniques- Status and Projections," *Computers and Structures*, Vol. 9, 1978, pp. 621-632.

48. Cox P., Burch R., Hocevar and Yang P., "SUPPLE: Simulator Utilizing Parallel Processing and Latency Exploitation," *Proc. of Int. Conf. on Computer-Aided Design*, Santa Clara, CA, November 1987, pp. 368-371.

49. Yuan C. P., Lucas R., and Chan P., Dutton R., "Parallel Electronic Circuit Simulation on the iPSC System," *IEEE 1988 Custom Integrated Circuit Conference*, Rochester NY, May 1988.

50. Sarje A. K. and Sagar G., "Hueristic Model for Task Allocation in Distributed Computer Systems," *IEE Proceedings-E* Vol. 138, No. 5, September 1991, pp. 313-317.

51. Efe K., "Heuristic Models of Task Assignment Scheduling in Distributed Computer Systems," *Computer*, 1982, Vol. 15, No. 6, pp 50-56.

52. Reed Daniel A. and Fujimoto Richard M., *Multicomputer Networks: Message-Based Parallel Processing*, MIT Press series in scientific computation, 1987.

53. Sadayappan P. and Ercal Fiket, "Cluster-partitioning Approaches to Mapping Programs onto a Hypercube," *Lecture notes in Computer Science*, 297, Springer-Verlag.

54. Stone H. S., "Multiprocessor Scheduling With the Aid of Network Algorithms," *IEEE Transations on Software Engineering*, SE-3, No.1, January 1977.

55. Stone H. S., Rao G. and Hu T. C., "Assignment of Tasks in a Distributed Processor System with Limited Memory," *IEEE Transactions on Computers*, Vol C-28, April 1979, pp. 291-298.

56. Bokhari S. H., "Partitioning Problem in Parallel, Pipelined and Distributed Computing," *IEEE Transactions on Computers*, Vol. 37, No. 1, 1988, pp. 48-57.

57. Bokhari S. H., "A Shortest Tree Algorithm for Optimal Assignment Across Space and Time in a Distributed Processor System," *IEEE Transactions on Software Engineering,* Vol, SE-7, No. 11, pp. 583-589.

58. Ma P. R, Lee E. Y., and Tsuchiya M., "A Task Allocation Model for Distributed Computing systems," *IEEE Transactions on Computers*, Vol. C-31, January 1982, pp. 41-47.

59. Ford L. R. and Fulkerson D. R., *Flows in Networks*, Princeton NJ, Princeton Univ. Press, 1962.

60. Lo Virginia Mary, "Heuristic Algorithms for Task Assignments in Distributed Systems," *IEEE Transactions on Computers*, Vol. 37, No. 11, November 1988, pp. 1385-1397.

61. Shield J., "Partitioning Concurrent VLSI Simulation Programs onto a Multiprocessor by Simulated Annealing," *IEE Proceedings-E*, Vol. 134, No. 1, pp.24-30.

62. Muhlenbein H., Groges-Schleuter and Kramer O., "New Solutions to Mapping Problem of Parallel Systems: The Evolution Approach," *Parallel Computing* 4, 1987, 269-279.

63. Gylys V. B. and Edwards J. A., "Optimum Partitioning of Workload for Distributed Systems," Digest of Papers, COMPCON 76, Fall 1976.

64. Chen Song and Eshaghian Mary M., "A Fast Recursive Mapping Algorithm," *Concurrency: Practice and Experience*, Vol. 7, No. 5, August 1995.

65. Bokhari Shahid H., "On Mapping Problem," *IEEE Transactions on Computers*, Vol. C-30, No. 3, March 1981.

66. Lee Soo-Young and Aggarwal J. K., "A Mapping Strategy for Parallel Processing," *IEEE Transactions on Computers*, Vol. C-36, No. 4, April 1987, pp. 433-441.

67. Sarkar V. *Partitioning and Scheduling Parallel Progrms for Execution on Multiprocessors*, MIT Press 1989.

68. Berman F.and Snyder L., "On Mapping Parallel Algorithms onto Parallel Architectures," *Journal of Parallel and Distributed Computing*, 4, 1987, 439-458.

69. *MCP 1000 Reference Manual*, Transtech Ltd, 1992.

70. *Occam2 Reference Manual*, Inmos Ltd, Prentice-Hall 1988.

71. Hoare C. A. R., *Communicating Sequential Processes*, Prentice-Hall, 1985.

72. Gerasoulis A. and Yang T., "A Comparison of Clustering Heuristics for Directed Acyclic Graphs on Multiprocessors," *Journal of Parallel and Distributed Computing,* Vol. 16, 1992, pp. 276-291.

73. *The Transputer Applications Notebook: Systems and Performance*, INMOS Ltd., 1989.

74. Xu J. and Hwang K. "Heuristic Methods for Dynamic Load Balancing in a Message-Passing Multicomputer," *Journal of Parallel and Distributed Computing*, Vol. 18, No. 1-13, 1993, pp. 1-13.

75. Lin, Frank C. H. and Kellor R. M.; "Gradient Models: A Demand-Driven Load Balancing Scheme," *IEEE 6th International Conference on Distributed Computing Systems*; May 1986.

76. Kale', L. V.; Comparing the Performance of Two Dynamic Load Distribution Methods; Report No. UIUCDCS-R87-1387, Dept. of Computer Science, University of Illinois; 1987.

77. D.M Nicol, P. F. Reynolds Jr, "Optimal dynamic remapping of data parallel computations," *IEEE Transactions on Computers*., Vol. 39, No. 2, (1990).

78. Xia E.Z. and Saleh R. A., "Parallel Waveform-Newton Algorithms for Circuit Simulation," *IEEE Transactions on Computer-aided Design of Integrated Circuits and Systems,* Vol. 11, No. 4, April 1992, pp. 432-442.

79. Smart David, *Parallel Processing Techniques for the Simulation of MOS VLSI Circuits using Waveform Relaxation*, Ph.D. dissertation, Dept. of Electrical Engineering, Univ. of Illinois at Urbana-Champagn, 1988.

80. Boreddy J. and Paulraj A., "On the Performance of Transputer Arrays for Dense Linear Systems," *Parallel Computing*, Vol. 15, 1990, pp. 107-117.

81. Zein David A., "Solution of a Set of Nonlinear Algebraic Equations for General Purpose CAD Programs," in *Circuit Analysis, Simulation and Design*, Elsevier Science Publishers B. V., 1986.

82. John Galletly, *Occam2*, Pitman Publishing, London, 1990.

A. DEVICE MODEL EVALUATION

Nodal analysis is a commonly used technique because it permits formulation of circuit equations by inspection for linear and non-linear circuits. The equation formulation process considers one device at a time. The device Branch Constitutive Equations (BCE) and node voltages are used to compute contributions of the device to the left and right hand side of the equation. This is known as device model evaluation. This appendix illustrates device model evaluation for linear and non-linear circuit elements with the help of simple examples. A pattern based technique described by McCalla [26] is presented.

A.1 Linear Devices

The nodal equations for a linear circuit can be expressed in the form:

$$YV = I, \qquad (a.1)$$

where Y is the nodal admittance matrix, V is the vector of node voltages to be found and I is the vector representing independent current sources. The term y_{ii} in Y represents the sum of admittances of all the branches connected to node i; y_{ij} in Y represents the negative sum of admittances of all branches connecting node i and node j. The term i_k represents the sum of source currents entering node k. If a resistor of value R is connected between nodes 3 and 5, $(G = 1/R)$ is added to y_{33} and y_{55} and subtracted from y_{35} and y_{53}. If a current source of magnitude I is connected between nodes 2 and 4 directed from 2 to 4. then I is subtracted from i_2 and added to i_4. The matrix representation of Equation a.1 with the resistor and current source is:

$$
\begin{array}{c}
\\
\varepsilon_1 \\ \varepsilon_2 \\ \varepsilon_3 \\ \varepsilon_4 \\ \varepsilon_5 \\ \varepsilon_6
\end{array}
\begin{array}{c}
\begin{array}{cccccc} 1 & 2 & 3 & 4 & 5 & 6\ . \end{array} \\
\left[\begin{array}{cccccc}
. & . & . & . & . & . \\
. & . & . & . & . & . \\
. & . & +G & . & -G & . \\
. & . & . & . & & . \\
. & . & -G & . & +G & . \\
. & . & . & . & & .
\end{array} \right]
\end{array}
\left[\begin{array}{c} v_1 \\ v_2 \\ v_3 \\ v_4 \\ v_5 \\ \ \end{array} \right]
=
\left[\begin{array}{c} \ \\ -I \\ \ \\ +I \\ \ \\ \ \end{array} \right]
$$

It is observed that each network element is associated with a distinct pattern. The pattern of a circuit element gives information about its contributions to the admittance matrix and RHS. For example a resistor pattern can represented as [26]:

	V +	V -	RHS
ε_+	1/R	-1/R	
ε_-	-1/R	1/R	

where *V+* represents the column in *Y* corresponding to the positive reference node and *V-* represents the column corresponding to the negative reference node. Similarly ε_+ and ε_- represent the rows of *Y* and the *RHS*. McCalla has discussed patterns of different network elements used for DC and transient analysis. Contributions made by a resistor to the admittance matrix are the same for DC and transient analysis.

Contributions made by a linear capacitor to the admittance matrix and RHS depend on the integration method used. Branch constitutive equations of a linear capacitor can be expressed as:

$$I = C\frac{dV}{dt}. \tag{a.3}$$

or

$$I_{n+1} = C\dot{V}_{n+1}. \tag{a.4}$$

Application of the Backward Euler Formula gives:

$$\begin{aligned} I_{n+1} &= \frac{C}{h}(V_{n+1} - V_n) \\ I_{n+1} &= \frac{C}{h}V_{n+1} - \frac{C}{h}V_n \\ I_{n+1} &= G_C V_{n+1} + I_C \end{aligned} \tag{a.5}$$

where $G_C = C/h$ and $I_C = -(C/h)V_n$. The resulting pattern of a capacitor is given by:

	V +	V -	RHS
ε_+	G_C	$-G_C$	$-I_C$
ε_-	$-G_C$	G_C	I_C

A.2 Nonlinear Devices

The nodal equations for a nonlinear circuit can be expressed in the form:

$$F(v) = 0, \tag{a.6}$$

A general form of the Newton-Raphson iteration equation to solve $F(v) = 0$, where $v \in R^N$ and $F: R^N \to R^N$ is:

$$J_F(v^k)(v^{k+1} - v^k) = -F(v^k), \tag{a.7}$$

where $J_F(v)$ is the Jacobian of $F(v)$ and v^{k+1}, v^k are $k+1$th and kth iterates respectively. The Jacobian can be expressed as:

$$J_F(v)=\begin{bmatrix} \left.\frac{\partial f_1}{\partial v_1}\right|_{v_k} & \left.\frac{\partial f_1}{\partial v_2}\right|_{v_k} & \bullet & \bullet & \left.\frac{\partial f_1}{\partial v_n}\right|_{v_k} \\ \bullet & & & & \\ \bullet & & & & \\ \left.\frac{\partial f_n}{\partial v_1}\right|_{v_k} & \left.\frac{\partial f_n}{\partial v_2}\right|_{v_k} & \bullet & \bullet & \left.\frac{\partial f_n}{\partial v_n}\right|_{v_k} \end{bmatrix} \qquad (a.8)$$

Model evaluation for a nonlinear device involves computing contributions of the device to the Jacobian and the RHS. This is explained with the help of an example of the Schichman-Hodges model of a MOSFET. This model has been described in detail in Chapter 2. Only those parts relevant to the discussion are given below.

Assume $B=\Gamma.W/L$ where Γ is a transconductance parameter and W/L is the width to length ratio. The drain-to-source current in the triode region is given by:

$$I_{ds}=B.V_{ds}(V_{gs}-V_T-1/2\,V_{ds})=f(V_g,V_s,V_d,V_x) \qquad V_{ds}<(V_{gs}-V_T) \qquad (a.9)$$

and the drain-to-source current in the saturation region is given by:

$$I_{ds}=1/2.B(V_{gs}-V_T)^2=f(V_g,V_s,V_x) \qquad V_{ds}\geq(V_{gs}-V_T) \qquad (a.10)$$

Also $I_{ds}=0 \quad if \quad (V_{gs}-V_T)\leq 0$ and the threshold voltage V_T can be expressed as:

$$V_T=V_{FB}+K(V_{ss}+\varphi)^{0.5} \qquad (a.11)$$

where V_{FB} is a flat-band voltage and K is a constant. Application of the Kirchoff's law at the d and s nodes gives:

$$\begin{aligned} f_d(v)&=J_{ds}+\cdots=0\\ f_s(v)&=-J_{ds}+\cdots=0 \end{aligned} \qquad (a.12)$$

The resulting pattern of matrix contributions of a FET is given by:

	s	g	d	x	RHS
s	Y_{ss}	Y_{sg}	Y_{sd}	Y_{sx}	$-f_s(v)$
d	Y_{ds}	Y_{dg}	Y_{dd}	Y_{dx}	$f_d(v)$

Assuming an N-channel device, the explicit form of the elements of the stamp for the saturation region is given below:

$$
\begin{aligned}
&Y_{ss} \equiv \frac{\partial I_{ds}}{\partial V_s} = -B(V_{gs} - V_T)[1 + 1/2K(V_{sx} + \varphi)^{-1/2}] \\
&Y_{sg} = B(V_{gs} - V_T) \\
&Y_{sd} = 0 \\
&Y_{sx} = -1/2B.K(V_T - V_{gs})(V_{sx} + \varphi)^{-1/2} \\
&RHS = -f_s(v) = -I_{ds} \\
&(Y_{ds}, Y_{dg}, Y_{dd}, Y_{dx}) \;\; = \;\; -(Y_{ss}, Y_{sg}, Y_{sd}, Y_{sx})
\end{aligned}
\qquad (a.14)
$$

In the triode region these contributions are:

$$
\begin{aligned}
&Y_{ss} = B\{(V_T - V_{gs}) - 1/2K.V_{ds}.(V_{sx} + \varphi)^{-1/2}\} \\
&Y_{sg} = B.V_{ds} \\
&Y_{sd} = B(V_{gs} - V_{ds} - V_T) \\
&Y_{sx} = 1/2B.K.V_{ds}.(V_{sx} + \varphi)^{-1/2} \\
&RHS = -f_s(v) = -I_{ds} \\
&(Y_{ds}, Y_{dg}, Y_{dd}, Y_{dx}) \;\; = \;\; -(Y_{ss}, Y_{sg}, Y_{sd}, Y_{sx}) \\
&RHS = f_d(v) = I_{ds}
\end{aligned}
\qquad (a.13)
$$

Some circuit simulators add a 10^{12} ohm resistor from source and drain nodes to ground to ensure non-zero contributions to the *s* and *d* rows. This large resistor has negligible impact on simulator accuracy. In addition, a *1* ohm resistor is also added in parallel, for the first few NR iterations to aid convergence [81]. This resistor can help convergence by giving a better initial estimate of the solution.

B. NUMERICAL ANALYSIS

This appendix presents a proof of the basic waveform relaxation theorem. It is based on the discussion by Debefve et al. [37]. A detailed discussion of the subject is given in [14]. The general form of the waveform relaxation algorithm may be described in its canonical form as:

$$\begin{aligned} \dot{x}^i &= f(x^i, x^{i-1}, \dot{x}^{i-1}, u) \\ x(0) &= x_0 \end{aligned} \qquad (b.1)$$

where f is a vector function which depends on the choice of the relaxation method used. The proof of the convergence of the WR theorem depends on the standard contraction mapping theorem which is stated below.

Contraction Mapping Theorem

Let Y be a Banach space and $F: Y \rightarrow Y$. If F satisfies $\|F(y) - F(x)\| \leq \gamma\|y - x\|$ for all $\gamma \in [0, 1)$ then f has a unique fixed point y which may be obtained from any initial guess $y^0 \in Y$ by a Picard iteration $y^i = F(y^{i-1})$.

Convergence Proof

The proof considers one iteration step and the convergence mapping principle is applied to Equation b.1. Variables $w = \dot{x}^{i-1}$, $z = \dot{x}^i$, and the operator S :

$$\begin{aligned} x^{i-1} &= x(0) + \int_0^t w(\tau)\, d\tau \equiv Sw \\ x^i &= x(0) + \int z(\tau)\, d\tau \equiv Sz \end{aligned}$$

are defined for notational convenience. Using this notation, Equation b.1 can be written as:

$$z = f(Sz, Sw, w; u).$$

The variable w refers to quantities from the previous iteration and the variable u refers to circuit input waveforms which are known. Therefore the equation above can be written as:

$$z = G(z)$$

It is necessary to show that the operator G is a contraction map in some norm. Consider the space of continuous functions on $[0, t{:}il.Convergence]$ with the norm

$$\|w\| = \sup_{t\in[0,T]} e^{-\lambda t}|w(t)| \tag{b.2}$$

where λ is a parameter to be chosen later. Assume that f in Equation b.1 is Lipshitz with constants K_1, K_2 with respect to first two arguments and contractive, with constant $\gamma < 1$, with respect to the third argument. Therefore

$$\begin{aligned}
z_1 &= f(Sz_1, Sw_1, w_1, u) \\
z_2 &= f(Sz_2, Sw_2, w_2, u) \\
z_1 - z_2 &= f(Sz_1, Sw_1, w_1, u) - f(Sz_2, Sw_2, w_2, u) \\
\|z_1 - z_2\| &= \|f(Sz_1, Sw_1, w_1, u) - f(Sz_2, Sw_2, w_2, u)\|
\end{aligned}$$

Using the Lipshitz condition gives

$$\|z_1 - z_2\| \le K_1\|S(z_1 - z_2)\| + \|S(w_1 - w_2)\| + \gamma\|w_1 - w_2\| \tag{b.3}$$

Consider the term $\|S(w_1 - w_2)\|$

$$\|S(w_1 - w_2)\| = \sup_{t\in[0,t]} e^{-\lambda t}|Sw_1(t) - Sw_2(t)|$$

As $\left|\int_0^t w(\tau)d\tau\right| \le \int_0^t |w(\tau)|\,d\tau$

$$\begin{aligned}
\|S(w_1 - w_2)\| &\le \sup_{t\in[0,t]} e^{-\lambda t}\int_0^t |w_1(\tau) - w_2(\tau)|\,d\tau \\
\|S(w_1 - w_2)\| &\le \sup_{t\in[0,t]} e^{-\lambda t}\int_0^t e^{\lambda\tau}.e^{-\lambda\tau}|w_1(\tau) - w_2(\tau)|\,d\tau \\
\|S(w_1 - w_2)\| &\le \|w_1 - w_2\| \sup_{t\in[0,t]} e^{-\lambda t}\int_0^t e^{\lambda\tau}d\tau. \\
\|S(w_1 - w_2)\| &\le \|w_1 - w_2\|\frac{(1 - e^{-\lambda t})}{\lambda} \le \frac{1}{\lambda}\|w_1 - w_2\|
\end{aligned}$$

Similarly $\|S(z_1 - z_2)\| \leq \frac{1}{\lambda}\|z_1 - z_2\|$

Substituting these results in Equation b.3 gives

$$\|z_1 - z_2\| \leq (r + K_2/\lambda)(1 - K_1/\lambda)^{-1}\|w_1 - w_2\|$$

Taking λ large enough one has

$$\|z_1 - z_2\| \leq \gamma\|w_1 - w_2\| \qquad \gamma < 1$$

Thus the operator G is a contraction map in the norm defined by Equation b.2 which proves the iteration defined by Equation b.1 converges to a unique fixed point. The continuity of G shows this fixed point satisfies the equation $z = Gz$.

C. TRANSPUTER AND OCCAM

Transputers are a family of processors designed for parallel processing [20]. Occam is a distributed memory parallel programming language specifically designed for Transputers [70][82]. It is based on C. A. R. Hoare's theoretical model of Communicating Sequential Processes (CSP) [71]. Within the CSP framework, a program is a collection of sequential processes each of which may be executing concurrently with others. The processes interact only via synchronized inter-process input/output operations. When a sender/receiver reaches an input/output operation it waits for the corresponding process to reach the matching operation. At this point the input/output operation is performed. There is no buffering or queuing of messages.

The Transputer family consists of several processors (e.g., T414, T800, T805 and T9000). The multiprocessor system used for this research uses the INMOS transputer T800 [20]. It is described below. The IMS T800 is a 32-bit microprocessor with a 64-bit floating point unit. The microprocessor runs at 20 MHz and can deliver 10 MIPS. The floating point unit operates concurrently with the microprocessor. It provides both single and double length operations. The unit implements IEEE floating-point arithmetic and can deliver 1.5 MFLOPS. The T800 contains a micro-coded priority scheduler and can time-share any number of concurrent processes. The context switching overhead is on the order of a few microseconds.

The processor has 4 KBytes of on-chip memory. It can directly address a memory address space up to 4 GBytes. Memory above the on-chip 4 Kbytes is accessed via the external memory interface. The IMS T800 has four high speed communication links. Each link can transfer data at over 1 Mbytes per second with automatic handshaking in each direction. The four communication links permit development of multi-transputer systems with different inter-connection network topologies such as pipeline, ring and hyper cube. The processor has two timers each for a priority level. The high

priority timer is incremented every microsecond and cycles approximately every 4,295 seconds. The low priority timer increments every 64 microseconds and cycles approximately every 76 hours.

An informal description of important aspects of Occam is given below [70]. An Occam program is a collection of communicating processes. Parallel processes communicate exclusively via Occam channels. The use of shared/common variables is not permitted. The fundamental unit of an Occam program is a primitive process. Primitive processes defined in Occam are assignment, input, output, SKIP and STOP. Assignment processes assign values to a named program variable. They have the form:

variable := expression.

The input process allows a value to be input from an Occam channel and assigned to a named variable. Input processes have the form:

channel ? variable

The output process outputs the value of an expression along a named channel. It has the form:

channel ! expression

The SKIP process starts, performs no function and terminates immediately. The STOP process starts and does not terminate. Program execution is held up after a STOP. It can be used to react to an illegal condition in the logic of the program.

Occam is a strongly typed language. The primitive data types available in Occam 2 are BYTE, INT16, INT32, INT64, REAL32, REAL64 and BOOL. The only structured data type defined in Occam is a multi-dimensional array. Channels used by a process are specified in the same way as variables and constants. A channel is associated with a protocol. The protocol defines the type of data to be transferred along the channel in any communication. A channel communication has the form:

CHAN OF type channel:

where type is the data type of the channel, and channel is the Occam identifier of the channel.

High-level processes may be built from primitive processes using Occam constructions. A construction consists of a collection of component processes. It becomes another Occam process. Important constructions are the sequential construction (SEQ), the parallel construction (PAR), and the alternate construction (ALT). The sequential construction causes component processes to be executed one after the other. It has the form:

```
SEQ
  process 1
      .
      .
  process n
```

The parallel construction causes processes to be executed concurrently, each at its own rate. The parallel construction terminates only after all the component processes have terminated. The parallel construction has the form:

```
PAR
  process 1
      .
      .
  process n
```

The alternate construction allows selection of a process from a list of component processes depending on the condition of the corresponding input guard. An alternate process terminates after execution of the selected process. It has the form:

ALT
input 1
process 1
.
.
input 2
process n

An important enhancement of the ALT construction is the PRI ALT (priority alternate) construction. If inputs are available on multiple channels, then this construction selects the process with the highest priority input guard. The first alternative process is the high priority process.

A few other important constructions are conditional construction, selection construction and repetition construction. A conditional construction has the form:

IF
Boolean expression 1
process 1
.
Boolean expression n
process n

A selection construction has the form:

CASE selector
case expression 1
process 1
.
case expression n
process n

The repetition construction has the form:

WHILE Boolean expression
process

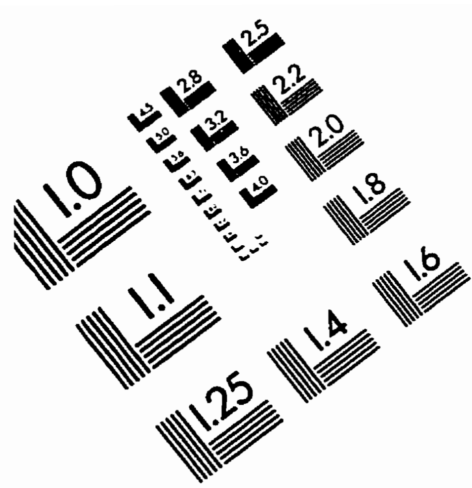
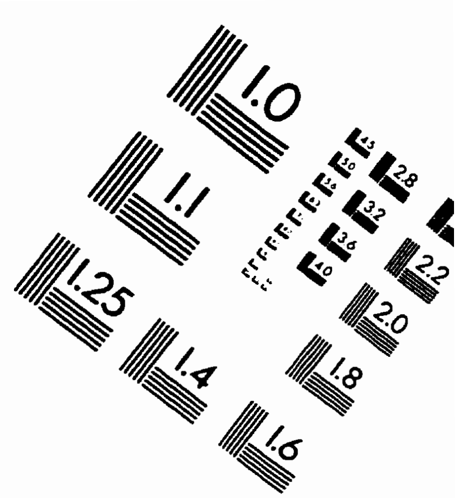
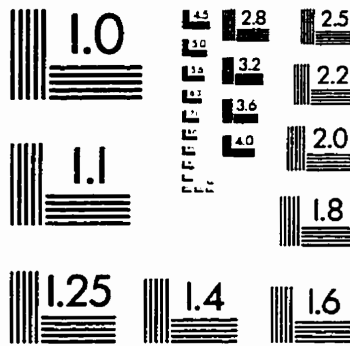
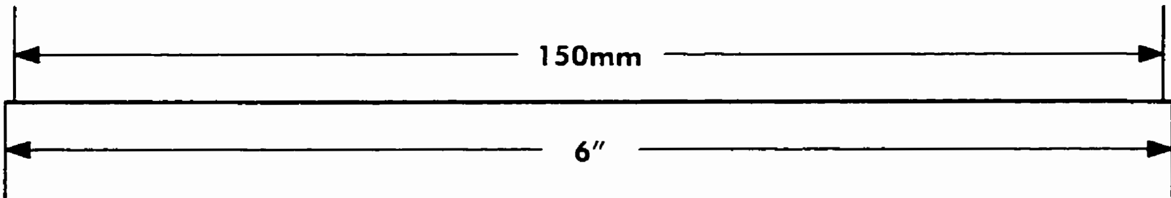
Occam has a versatile replicator feature. It has the form:

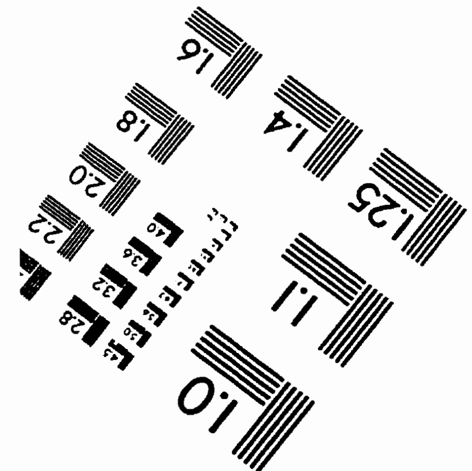
index = start FOR count

It is possible to replicate SEQ, PAR, IF and ALT constructions.

Parallel programs in Occam are usually developed and tested on a single Transputer. It is possible to distribute the same program on multiple Transputers without any changes in software. Occam provides mechanisms to specify the configuration of a program. Configuration associates the components of an Occam program with a set of physical resources.

IMAGE EVALUATION
TEST TARGET (QA-3)



APPLIED IMAGE, Inc
1653 East Main Street
Rochester, NY 14609 USA
Phone: 716/482-0300
Fax: 716/288-5989

© 1993, Applied Image, Inc., All Rights Reserved