DIRECT METHODS IN CIRCUIT SIMULATION USING MULTIPROCESSORS

by

George K. Jacob

Memorandum No. UCB/ERL M87/67

7 October 1987

DIRECT METHODS IN CIRCUIT SIMULATION USING MULTIPROCESSORS

by

George K. Jacob

Memorandum No. UCB/ERL M87/67

7 October 1987

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

DIRECT METHODS IN CIRCUIT SIMULATION USING MULTIPROCESSORS

by

George K. Jacob

Memorandum No. UCB/ERL M87/67

7 October 1987

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

DIRECT METHODS IN CIRCUIT SIMULATION USING MULTIPROCESSORS

George K. Jacob

Ph.D.

Department of Electrical Engineering
and Computer Science

Sponsors: DARPA, SRC, Cal State Micro

Signature 

Donald O. Pederson
Committee Chairman

ABSTRACT

Circuit simulation, the solution of a set of nonlinear ordinary differential equations (ODE's) that describe a circuit's behaviour, continues to be an important tool for the design of integrated circuits (IC's). However, circuit simulation speeds have not kept pace with the increasing number of transistors and complexity of IC's. Parallel processing, the technique of partitioning a large problem into sub-problems and solving the sub-problems simultaneously, is a potential method of improving the speed performance of circuit simulators. This dissertation addresses the issue of applying parallel-processing methods to reduce circuit-simulation runtimes.

The nonlinear ODE's in circuit simulation are commonly solved using either direct or mixed direct-relaxation methods. Since relaxation methods provide an implicit partitioning scheme which makes parallel processing easy, the performance of a node-based relaxation circuit simulator implemented on the BBN Butterfly Parallel Processor is studied and indicates that relaxation-based circuit simulation is an application well-suited to parallel processing. However, relaxation methods alone are not appropriate for the simulation of all

circuits and direct methods continue to be important.

The parallelization of a direct-method circuit simulator is studied on the Sequent Balance B8000 parallel processor. The major time-consuming phases of direct-method circuit simulation, model evaluation and linear-equation solution, are evaluated for their suitability to parallel processing. Model-evaluation time is observed to decrease almost linearly with increasing numbers of processors, proving it is well-suited to parallel processing. Linear-equation solution and synchronization points between the two phases are identified as obstacles to the efficient parallelization of direct-method simulators. A new pipelined direct-method algorithm is presented that eliminates the synchronization point between model evaluation and linear-equation solution, thereby improving parallel performance.

ACKNOWLEDGEMENTS

This dissertation has been made possible through the efforts of a number of people, although attributed to but one. The following acknowledgements are an attempt to express my immeasurable gratitude to those without whom this document would not have been. To the numerous unmentioned due to constraints of space, time and, alas, memory, my apologies and silent thanks.

Prof. Pederson, my thesis advisor, has been an able mentor and guide, providing advice when I needed it and allowing me (or goading me, if the situation demanded!) to stand on my feet when this was important. Prof. Newton has, through his constant demand for excellence, pushed me to the conclusion of tasks that I might have been wont to leave half-done. Beyond the guidance towards my research and dissertation, I have received from my advisors presentation skills, an appreciation for good work and a sense of perspective that I shall always value. This dissertation serves as adequate testament. I am grateful to Prof. Sangiovanni-Vincentelli for his support, and along with Profs. Pederson and Newton, for the excellent CAD-group environment they have built at Berkeley. The financial support of the California State MICRO program, the Semiconductor Research Corporation and DARPA are gratefully acknowledged.

I thank Prof. Jewell for his support and for reviewing this dissertation. I also thank Profs. Randy Katz, Larry Carter and Jeanne Ferrante for their support of my work. I am grateful to a number of people at Bolt, Beranek and Newman, especially John Goodhue, Bob Thomas, Walter Milliken and Will Crowther, for introducing me (gently) to the wonderful world of multiprocessing, and to Jon Payne for introducing me to Jove. I am also grateful to Tony Magpayo and the folks at Sequent for putting up with the many questions I had about the Balance.

Karti Mayaram, Theo Kelessoglou, Wayne Christopher, Jeff Burns, Srinivas Devadas, Lorraine Layer, Res Saleh, Ron Gyurcsik and Ileana Ocneanu have been constant sources of

information, good cheer, humour and inspiration and I thank them for their steady support and wonderful company. I am grateful to Don Webber, Young Kim, Seung Hwang, Ken Kundert, Shelley Sprandel, Tom Boot and everyone in the CAD group for assistance at times when I didn't have the foggiest idea of what was happening to the world around me, and to Randy Cieslak, Tim Salcudean, Myra Boenke, Ted Baker, Güntekin Kabuli, Barbara Mills, John Kunze, Roger Hale, and Herve' and Annick Da Costa for being around to talk to and divert my attention when I found out. I am also grateful to the many who permitted me to vent my frustrations ("research-derived?"; "nay, sir, surely ye jest") on the squash or tennis court.

I thank my room-mates, Amit Bhaya, Aniruddha Das, Nazli Gündeş and Sunil Kumar for putting up with and without me, and Vidula Kirtikar, Susan Chacko, and Deb, Bruce and Stephen Smith for their constant love and support. Most of all, I am grateful to my brothers, Jonathan and Rahul, for keeping me humble yet letting me believe I knew who was boss, and to my parents, Jake and Sheila, for picking me off the street.

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

Simulators are used extensively by Very Large Scale Integrated (VLSI) circuit designers in order to study circuit behaviour prior to fabrication. An accurate circuit simulator reads an input description detailing a circuit's connectivity and element parameters and assembles a set of non-linear ordinary differential equations (ODE's) to represent the circuit behaviour. For a prescribed set of input signals over a desired time period, a circuit simulator numerically solves the non-linear ODE's to yield the simulated circuit behaviour, typically presented graphically as output-signal waveforms. Circuit designers study the output waveforms and then modify circuit elements or connectivity until the output signals meet the required design specifications.

Fast circuit simulation is important for the design of electronic circuits, and reliance on circuit simulators is expected to continue as circuit designs and device technologies become more sophisticated. Today's commonly used circuit simulators, e.g., ADVICE, ASTAP, SLATE and SPICE [1,2,3,4] which use direct methods to solve the circuit equations, and SAMSON, SPLICE and RELAX [5,6,7,8,9,10] which use mixed direct-relaxation techniques, are not able to simulate 100,000-transistor VLSI circuits within a reasonable time and are growing increasingly inadequate as part of a circuit designer's design-simulate-redesign cycle, especially for certain classes of circuits, e.g., mixed analog-digital circuits and dynamic memories. As a result, significant research is being conducted towards building faster circuit simulators using hardware accelerators [11,12,13,14,15,16,17,18].

Previous techniques for speeding up circuit simulators include generation of special-purpose microcode for linear-equation solution [19], software [6,20,21,22] and hardware [23] table-lookup for empirical model evaluation, and the use of vector processors [24,25] such as the CRAY-1 [26] and array processors [27,28] such as the FPS-164 [29]. For practical circuits, the overall speed improvement of the simulation has been well under two orders of magnitude compared to SPICE2G [30]. The hardware-related studies [23,24,25,28] are particularly interesting as they show that *a significant amount of parallelism is present* during the linearization of the circuit matrix.

In [24], it was shown that the sparsity and irregular structure of the circuit matrix caused the data gather-scatter time to dominate simulation time on vector processors. A dominant *gather-scatter* time implies that simply fetching and storing data from and to memory take longer than the actual computation using the data. More recently, vector processors such as the CRAY X-MP, the Hitachi S-810 and the CDC CYBER-205 [31,32,33] have been developed with special-purpose hardware for gather-scatter operations and have been utilized to overcome the gather-scatter problem in circuit simulation [34]. However, the Single Instruction Multiple Data (SIMD) operation mode of vector processors [35] results in inefficient utilization of processors during model evaluation, since differing regions of transistor operation necessitate program branches [36]. Thus, efficient parallel techniques must exploit the inherent parallelism of circuit simulation while simultaneously keeping as many processors busy as possible.

Parallel processing, the technique of decomposing a large problem into a number of (smaller) subproblems that are solved by a number of processors in parallel, provides one effective solution to both, the gather-scatter and the SIMD problems. The gather-scatter

bottleneck is avoided because distributed processors access distributed data, rather than initially distributed data being gathered to and scattered from centralized processors (as is the case with vector processors). Since parallel processors operate in Multiple Instruction Multiple Data (MIMD) [35] mode, processors can simultaneously evaluate transistors that are in different regions of operation, thereby exploiting more parallelism than can the SIMD vector processors. As a result, parallel processing appears well-equipped to improve the speed performance of circuit simulators. Parallel processing is made even more attractive by the recent emergence of a number of commercially viable multiprocessors that are built using inexpensive microprocessors, such as the BBN Butterfly Parallel Processor, the Intel iPSC and the Sequent Balance B8000 [37,38,39] or special-purpose IC chips for parallel processing, as in the Alliant and ELXSI machines [40,41]. The introduction of these multiprocessors also makes available a number of different parallel processor architectures, providing users with an opportunity to examine the fit between particular applications and the different architectures.

Efficient parallel processing for speedup requires the selection of an algorithm that displays a high degree of parallelism. Today's circuit simulators use either direct or mixed relaxation-direct algorithms. Direct-method simulators solve all the circuit equations together simultaneously (as a single vector system), including those describing inactive parts of a circuit, and have not proved efficient for rapid simulation of large digital circuits with high levels of latency. Latency of a circuit is defined in terms of both space and time: latency in space refers to the signals in the circuit that are inactive at a given timepoint during a simulation, while latency of a signal in time indicates those timepoints when the signal is inactive during the simulation period. Mixed relaxation-direct circuit simulators decompose a circuit into subcircuits, solving *active* subcircuits using direct

methods and iterating between subcircuits, using relaxation, until they converge to a correct solution [42]. Relaxation simulators initially appear better suited to parallel processing than do direct-method simulators, as they naturally partition a circuit into subcircuits that can be solved independently in parallel. However, since subcircuits are assembled on the basis of coupling strength between circuit elements, there is often a large variance in subcircuit size. As a result, balancing load between processors is difficult because processors assigned small subcircuits become idle while other processors continue to work on large subcircuits [10,43]. Thus, for efficient load balancing, the (direct-method) subcircuit solutions themselves must be sub-divided into smaller subtasks which, in turn, can be processed in parallel. Also, direct methods continue to be important for the simulation of tightly coupled circuits, as well as for dc analysis and for circuits that contain elements that are not easily handled by relaxation methods. The parallel potential of direct-method circuit simulation is investigated in this dissertation, through the use of the direct-method subcircuit solver in the parallel Iterated Timing Analysis (ITA) relaxation simulator, PSPLICE3 [43]. In order to distinguish the simulation program developed and described in this dissertation from the unparallelized subcircuit solver in PSPLICE3, the program developed here and listed in Appendix A is referred to as PDSPLICE3. PDSPLICE3 includes both, the parallel relaxation algorithms installed by Res Saleh in PSPLICE3, as well as the parallel direct-method algorithms described in this dissertation.

An introduction to direct-method circuit simulation techniques is presented in Chapter 2. Previous work towards accelerating direct-method circuit simulation through the use of parallel architectures is also described in Chapter 2. This presentation of uniprocessor, direct-method algorithms and previous parallelization experiments provides background as well as direction for the current study.

Experiments to increase the efficiency of a multiprocessor-based relaxation circuit simulator, MSPLICE, are described in Chapter 3. This description presents useful guidelines for programming a large multiprocessor system, hence the parallel programming techniques used for the MSPLICE experiment are applied to the parallelization of direct methods. In addition, MSPLICE's performance indicates that node-based relaxation simulators are highly parallelizable, as has been verified in [44]. The efficient parallelization of MSPLICE also helps to identify load balancing as the cause for the inadequate parallel performance of subcircuit-based relaxation simulators as in [10,43].

The first step towards parallelizing an application is the straightforward development of a parallel algorithm based on the best sequential algorithm for the application. Chapter 4 describes a parallel implementation of the most commonly used direct-method circuit-simulation algorithm on the Sequent bus-based Balance multiprocessor. The two most computationally expensive phases of direct-method simulation are the steps of the Newton-Raphson iterative loop, i.e., linearization and linear-equation solution, which are focussed on in this chapter. The linearization phase is demonstrated to lend itself easily to parallelization, with minor task-clumping modifications for efficiency on the Balance. A dataflow method using pivot-dependence graphs (PDG's) similar to those presented in [45] is utilized as a straightforward means of parallelizing sparse linear-equation solution, but it is shown to yield a small degree of parallelism which is difficult to realize on the Sequent Balance.

With parallelization of sparse linear-equation solution identified as a bottleneck in the performance of a parallel direct-method circuit simulator, further approaches to parallelizing linear-equation solution are described in Chapter 5. Theoretical limits to the parallel-

ism available using different algorithms are calculated, displaying that the PDG technique provides a larger degree of parallelism as circuit size increases. Further, row-based techniques are shown to have greater parallel potential than pivot-based methods, especially as the density of the circuit matrix increases. A row-dependence graph (RDG) algorithm is presented that efficiently exploits the limited parallelism available during sparse linear-equation solution and results are presented for an implementation on the Sequent Balance.

In spite of the efficient exploitation of parallelism in both the linearization and linear-equation solution phases, however, it is evident that the overall speedup for simulation is still limited by the low degree of parallelism during linear-equation solution and by synchronization bottlenecks between the phases. Removal of inter-phase bottlenecks by the use of pipelining improves the potential for parallelizing circuit simulation as a whole. In Chapter 6, a global parallelization scheme that pipelines the highly parallel linearization with the less-parallel linear-equation solution is presented. Pipelining simultaneously removes the synchronization point between the two phases of the Newton-Raphson method and improves the overall parallel potential. In addition, algorithms for parallel convergence checking and for pipelining back-substitution (during linear-equation solution) and convergence checking are presented, which lead to improvements in parallel performance for the direct-method solver as well as for the parallel relaxation simulator that utilizes the direct-method solver only for subcircuit evaluations.

The work described in this dissertation provides insights into parallelizing direct-method circuit simulation. The major conclusions of this study and possible directions for future research are listed in Chapter 7.

CHAPTER 2

DIRECT-METHOD CIRCUIT SIMULATION

Transient analysis, one of the most commonly used tools in circuit design [6, 19, 24], involves the solution of a system of nonlinear ordinary differential equations (ODE's) that describe the dynamic behaviour of a circuit. These equations may be written in the form

$$F(\dot{x}, x, t) = u(t), \quad x(0) = X_0 \tag{2.1}$$

where x is the vector of circuit variables, F represents a mapping function, $u(t)$ is the vector of input variables at time t, and X_0 represents the values of the circuit variables at time, $t = 0$ [46]. Techniques for constructing and solving Equation 2.1 using direct methods are described in Section 2.1. In Section 2.2, a number of hardware-based approaches to enhance the speed performance of direct-method circuit simulators are described and evaluated, providing a basis and direction for this study.

2.1: Algorithmic Techniques for Direct-Method Circuit Simulation

A complete representation of a circuit uses Kirchchoff's Voltage Law (KVL), Kirchchoff's Current Law (KCL) and branch equations for each element in the circuit, assembled as the Sparse Tableau [47] shown in Equation 2.2:

$$\begin{bmatrix} A & 0 & 0 \\ 0 & I & -A^t \\ B_i & B_v & 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ S \end{bmatrix} \tag{2.2}$$

where A is the reduced incidence matrix, B_i and B_v are submatrices that contain branch equation coefficients for current and voltage, respectively, and S is the vector of independent source currents and voltages. Programs, such as ASTAP [2], solve this system using

Sparse Tableau Analysis [48]. However, a sparse-tableau representation contains $n+2b$ equations, where n is the number of nodes and b is the number of branches in the circuit, while other techniques result in far smaller systems, e.g., Nodal Analysis has only n equations. Because of their compactness, nodal and loop analysis techniques, presented in Section 2.1.1, are better suited for computers with limited memory sizes. As a result, most circuit simulators use Modified Nodal Analysis [49], a mixed nodal- and loop-analysis technique, which is described in Section 2.1.1.

Given the system of nonlinear ODE's describing the circuit behaviour, the simulation proceeds over the desired time period as follows (also shown pictorially in Figure 2.1):

(a) At each timepoint, the nonlinear ODE system is integrated numerically to obtain a nonlinear algebraic system that represents the circuit at that timepoint;

(b) The system of nonlinear algebraic equations are iteratively solved using the Newton-Raphson method, i.e.,

(i) the nonlinear equations are linearized about the old solution point;

(ii) the resulting sparse system of linear equations is solved using LU-decomposition, forward- and back-substitution and yields voltage and current values for a new solution point.

(iii) convergence is checked for by comparing the old and new solution points: if the two points are within allowed tolerances of each other, simulation proceeds at the next timepoint; else, the Newton-Raphson process is continued at the same timepoint.

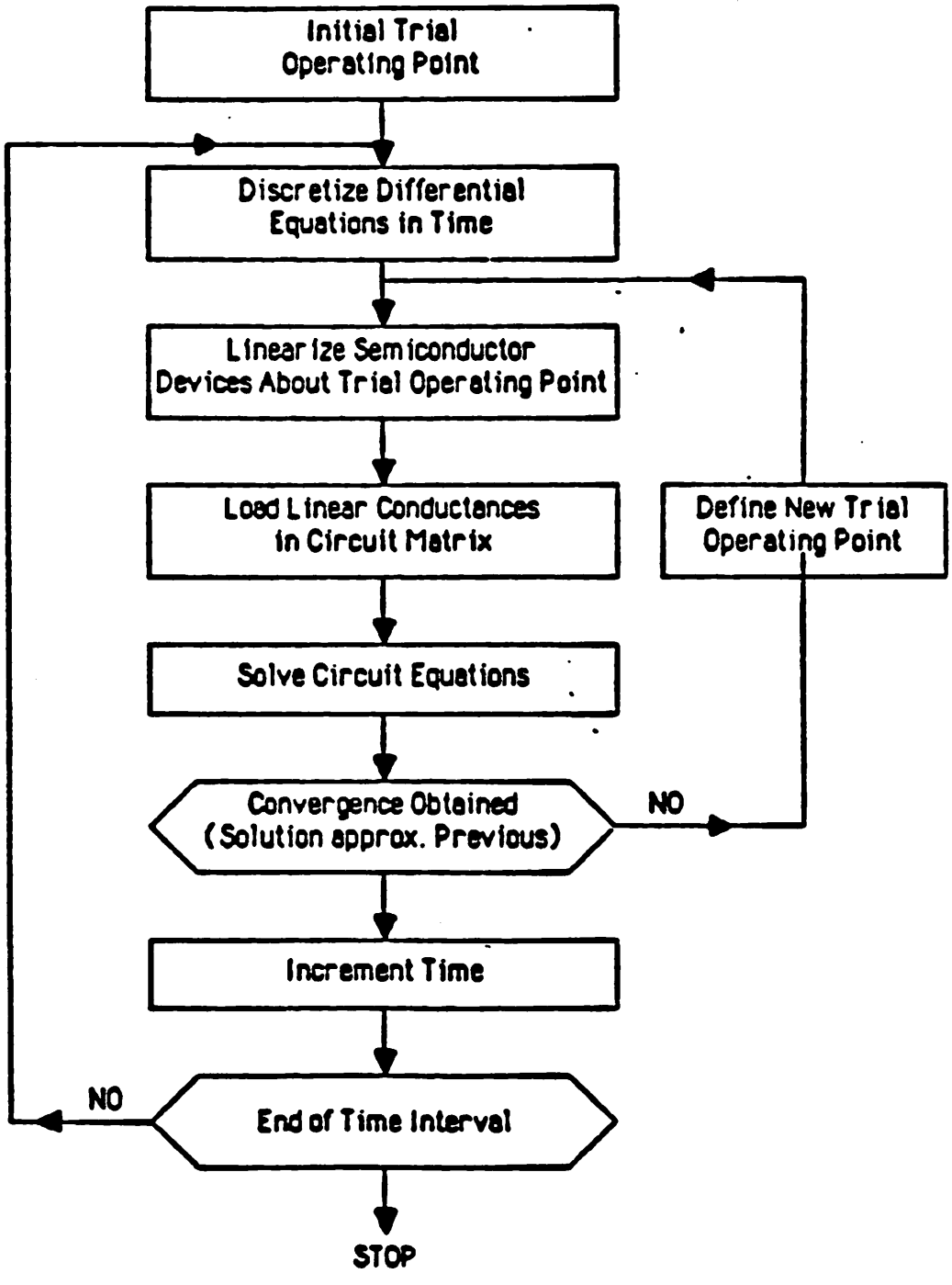

Figure 2.1. Standard Algorithm for Direct-Method Circuit Simulation

2.1.1 – Problem Construction

The two techniques commonly used to reduce the dimensions of the Sparse Tableau system of circuit equations are Nodal Analysis and Loop Analysis. *Nodal Analysis* [4,49] starts with the KCL equations at each circuit node, which equate the sum of all the branch currents entering each node to zero. Branch current-voltage relationships transform the KCL equations to a set of equations in terms of branch voltages, which are then converted using KVL to a system described by the circuit node voltages [47]. Since this system is over-determined, a single node is assigned as the datum node (at zero potential), and all other node voltages are calculated with respect to the datum node. The second technique, *Loop Analysis*, is analogous to Nodal Analysis, but starts with the KVL equations: the KVL equations are assembled for each loop in the circuit, and branch voltages are replaced using the branch relations and KCL, resulting in a system described by the loop currents.

VLSI circuit devices, e.g., diodes, bipolar and metal-oxide semiconductor (MOS) transistors, tend to be voltage-controlled and are therefore more easily represented by a technique that describes a circuit in terms of voltages, such as Nodal Analysis, than a representation in terms of currents, as in Loop Analysis. Hence, most circuit simulators use Nodal Analysis as a basis to assemble the circuit equations [50,4,5,7,10].

Nodal Analysis, however, is unable to handle either current-controlled devices or devices without current terms in their branch equations, e.g., independent voltage sources and voltage-controlled voltage sources, since branch relations for the former elements contain no voltage term while branch relations for the latter do not contain a term for current. In addition to the Nodal Analysis equations, these *non-nodal* elements require Loop Analysis equations of their branches. The resulting technique is known as *Modified Nodal*

Analysis [49] (MNA).

The MNA technique first uses Nodal Analysis to describe the circuit behaviour, ignoring currents for those branches with non-nodal elements. Then, branch equations for the non-nodal elements are added to describe the system completely. As a result, the final system to be solved has more than $n-1$ equations but is still significantly smaller than the sparse tableau. This is illustrated in Equation 2.3 below:

$$\begin{bmatrix} Y_n & A \\ B & C \end{bmatrix} \begin{bmatrix} e \\ i' \end{bmatrix} = \begin{bmatrix} I_s \\ E_s \end{bmatrix} \tag{2.3}$$

where Y_N is the $n-1 \times n-1$ nodal admittance matrix, I_s is the $n-1 \times 1$ vector of independent current sources, e is the $n-1 \times 1$ vector of node voltages, A is an $n-1 \times y$ matrix representing the effect of the y irreducible branch currents in nodal analysis, B and C are $y \times n-1$ and $y \times y$ matrices that give the branch equations for the irreducible branches, i' is the $y \times 1$ vector of irreducible branch currents, and E_s is the $y \times 1$ vector of independent voltages sources in the irreducible branches. Circuit simulators use numerical techniques to solve Equation 2.3.

2.1.2 – Integration Schemes

Numerical integration is employed to convert the nonlinear ODE system to a set of nonlinear algebraic equations. For circuit simulation, the numerical integration techniques used are *linear, multistep methods*, which divide the simulation interval into a number of timesteps [51]. Then, for each timestep, the system's behaviour is approximated by a linear function of past and present voltages, currents and charges. Thus, a linear, multistep method starts from the known dc solution for the circuit, calculates the circuit variables at the end of the first timestep and continues this process till the end of the simulation period,

while limiting the error at each step. The procedure of numerical integration is illustrated below using a single ODE for simplicity.

Given a first-order ODE of the following form:

$$\dot{y} = f(y,t)$$

linear, multistep methods divide the simulation period, T, into a number, n, of periods, t_n, such that:

$$t_0 = 0, \cdots, t_{n+1} = t_n + h_{n+1}, \cdots, t_N = T, n = 0,1,\ldots,N$$

where h_n is the timestep taken to arrive at the end of the nth period. Multistep methods compute y_{n+1} on the basis of y and $\dot{y}$ at p previous time points, in the following form:

$$y_{n+1} = \sum_{i=0}^{p} a_i\, y_{n-i} + \sum_{i=-1}^{p} h_{n-i}\, b_i\, \dot{y}_{n-i} \tag{2.4}$$

An example of an explicit method, where $b_{-1} = 0$, is the *Forward Euler (FE)* method, which is described by

$$y_{n+1} = y_n + h_{n+1}\, \dot{y}_n \tag{2.5}$$

Note that the FE method, as with all explicit methods, requires merely a function evaluation in Equation 2.5. Implicit methods, like the commonly used Backward Euler and Trapezoidal methods, have $b_{-1} \neq 0$ and require iterative solutions. For example, the *Backward Euler (BE)* method is described as

$$y_{n+1} = y_n + h_{n+1}\dot{y}_{n+1} \tag{2.6}$$

while the *Trapezoidal (TR)* method can be represented as

$$y_{n+1} = y_n + (h_{n+1}/2)\,(\dot{y}_{n+1} + \dot{y}_n) \tag{2.7}$$

Implicit methods, due to the "feedback loop" in their formulation, are generally more stable than explicit methods and provide error control and are hence preferred for integration.

Since an integration method approximates a continuous function by a linear equivalent, an error may be introduced due to truncation of the function. At each timestep, the truncation error due to a multistep method is dependent on the errors in the values calculated at the previous steps and is termed the *Global Truncation Error*. However, it is possible to compare the accuracy of integration schemes in an approximate manner merely by comparing their respective *Local Truncation Errors (LTE's)*, where the LTE is defined as the error between the calculated and actual values of the function at a given timestep, assuming that solutions at all previous timesteps were exact. The LTE of an integration method is a function of the timestep, the coefficients of the method, and the order of the method (determined by the polynomial of the highest degree exactly solvable by the method). Therefore, circuit simulators control the LTE at each timestep by controlling the size of the timestep. In the limit, as the timestep tends to zero, the error tends to zero as well; however, this increases the number of timesteps taken during the simulation period. As a result, circuit simulators take the largest timestep possible to keep the simulation time small, yet ensure that the LTE at each node in the circuit is always less than a certain pre-assigned constant.

In spite of restricted LTE's at each timepoint, an integration method may sometimes be unstable, i.e., its use results in predictions of instability for stable systems. It can be shown [46] that the FE method is stable for only a certain range of timesteps; the BE method is over-stable, i.e., it will make unstable systems appear to be stable, thereby quelling real oscillations; and the TR method is A-stable, i.e., it represents stable and unstable systems as being stable and unstable, respectively, regardless of the timestep used. The TR method is commonly used in circuit simulators.

Due to the many independent energy storage elements in a circuit, there is often more than a single time constant associated with a circuit. The chosen integration scheme should thus be *stiff* [46], i.e., it should be able to handle a wide range of time constants. If the integration method were only stable for small timesteps, as is the FE method, then the simulation would require many timesteps, even though the signals in the circuit may not be changing rapidly. A *stiffly stable* integration scheme, such as BE and TR, is stable for any size of timestep, and thereby facilitates the use of small timesteps when circuit signals are changing rapidly, or large timesteps when circuit activity is slow.

Once the nonlinear ODE's have been integrated numerically, they yield a set of non-linear algebraic equations. These equations are then solved directly, using the Newton-Raphson method.

2.1.3 – The Newton-Raphson Method

The Newton-Raphson (NR) method is an iterative technique used to solve a nonlinear equation. Each iteration proceeds as follows:

(a) Estimate an initial solution to the equation,

(b) Linearize the equation around the estimated solution, and

(c) Calculate the solution to the linear equation.

This process is repeated, using the solution from one iteration as the estimated initial solution for the next iteration, until the difference between the last two solutions is smaller than the weighted sum of an absolute tolerance and the product of the current solution with a relative tolerance, when the method is said to have *converged* to a solution.

2.1.3.1 – Linearization of the Circuit Matrix

At each iteration, i, the NR method involves determining a solution, x', to the following equation

$$x^{(i+1)} = x^i - J(x^i)^{-1} f(x^i) \tag{2.8}$$

where the elements of the Jacobian, $J(x^i)$, represent linearized circuit elements that would be obtained by linearizing each branch equation independently at x^i. Given that $J(x)$ is Lipschitz-continuous, i.e., $J(x)$ is continuous and its rate of change is bounded and that $J(x')$ is non-singular, the NR method converges provided the initial estimate is close to x'. Further, the NR method converges quadratically in the immediate vicinity of x', i.e., the error decreases quadratically with successive iterations. Hence, the NR method has a high rate of convergence when the estimated solution is close to the true solution, underlining the need for a good initial estimate.

When the estimated solution is distant from the true solution, the NR method does not converge rapidly and may not converge at all, e.g., when oscillations occur between two or more incorrect estimates. This problem may be addressed by methods that increase the region of convergence. For example, the source-stepping method allows sources to be activated in small steps, thereby reducing the distance between the initial estimate and the final solution.

While using the NR technique, it is possible to obtain some values of the local gradient (within the Jacobian matrix) that are high and predict unrealistic future values for certain circuit variables. On a computer, this results in numerical overflow and must be limited to allow the simulation to continue. The technique used in a number of circuit simulators to handle this problem is that of voltage and current *limiting*. Limiting schemes are

imposed only when predicted circuit variables, namely voltages and currents, rise above a certain threshold. In such cases, the highest-valued variables are reduced to a limiting value, and all other predicted variables are adjusted proportionately.

Since circuit model evaluation is expensive, a number of techniques have been employed to reduce the time taken for the linearization phase. It has been shown that table lookup reduces model-evaluation time significantly, by eliminating the need for repeated calculation of device-function coefficients [20,21,23,52]. Some circuit simulators save time by not updating the Jacobian matrix at each NR iteration, arguing that the Jacobian matrix is inaccurate when distant from the solution, while close to the solution the Jacobian matrix's accuracy makes frequent updates unnecessary [53,10]. Finally, it has been experimentally shown [6] that an *element-bypass* technique, where models are only evaluated if their input voltages and currents have changed by a significant amount, can result in significant savings, especially for predominantly digital circuits with large degrees of latency. If all elements are by-passed, and only the right-hand-side vector is updated, this approach is equivalent to the Jacobian bypass scheme.

2.1.3.2 – Sparse Linear-Equation Solution

Gaussian Elimination (GE) is one of the most efficient algorithms available to solve a system of linear equations represented as

$$Ax = b \tag{2.9}$$

where A is a non-singular $n \times n$ matrix, and x and b are n-dimensional vectors for the unknowns and right-hand side, respectively. LU-decomposition, a technique that uses the same number of operations as Gaussian Elimination, is commonly used in circuit simulators because, unlike GE, it does not necessitate modification of the right-hand side and is

hence useful for analyses where multiple right-hand sides are used. LU-decomposition involves decomposing the system in Equation 2.9 to

$$LUx = b \tag{2.10}$$

where L is a lower-triangular matrix and U is an upper-triangular matrix. Then, the system

$$Ly = b \tag{2.11}$$

is solved for the unknown y using *Forward Substitution*, following which the system

$$Ux = y \tag{2.12}$$

is solved for the unknown x by *Back Substitution*. LU-decomposition is attractive when the Jacobian matrix is not updated at each iteration, since the linear-equation solution involves only forward and back substitution and the decomposition need not be repeated until the Jacobian matrix is updated.

Circuit simulators employ strategies to maintain the sparsity that characterizes circuit matrices. Typically, a circuit matrix has three or four non-zero elements per row or column, usually such that the matrix has an irregular structure. The Markowitz technique [54], commonly used to determine pivoting order, assigns each pivot candidate a Markowitz row count and a Markowitz column count, which are the number of non-zeroes in the row or column other than the pivot candidate itself. Then, the pivot candidates are ordered to be eliminated in order of increasing Markowitz product (the product of the Markowitz row count and the Markowitz column count). For circuit simulation, the pivoting order is usually determined once each for dc solution and for transient analysis, although it is possible that circuit dynamics may necessitate re-ordering during a transient analysis. Circuit matrices are re-ordered during transient analysis only if the magnitude of one of the diagonal elements in the matrix falls below a certain threshold.

The example matrix shown in Figure 2.2(a) illustrates the ordering of a small matrix using the Markowitz criterion. In Figure 2.2(a), non-zeroes in the matrix are denoted by X's and zeroes are denoted by O's. With the configuration in Figure 2.2(a), after the first row has been eliminated, all the zeroes in the matrix are filled in, and the matrix is subsequently 100% dense, resulting in a total of 36 floating-point operations on the matrix. However, using the Markowitz criterion, the configuration in Figure 2.2(b) is arrived at, no fill-ins are created and a total of 16 floating-point operations are performed during the LU-decomposition.

The diagonal elements of the matrix A, known as *pivots* when selected during matrix ordering to stay on the diagonal, are crucial to the accuracy of linear-equation solution since the elements in a row are divided by the corresponding pivot element. Thus, if a pivot element is much smaller than the rest of the elements in the row, inaccuracies may occur due to computer finite-precision arithmetic. As a result, circuit simulators employ *pivoting strategies*, to determine which matrix elements should be used as pivots in order to ensure accuracy through the simulation [55].

2.2: Hardware-based Acceleration Techniques

While a number of algorithmic techniques [5,6,9,10,43,50,52] have led to electrical circuit simulation that is almost two orders of magnitude faster than SPICE, large-scale improvement in the speed performance of circuit simulators continues to depend on hardware-based approaches, presented in this section.

In Section 2.2.1, work on vector and array processors is described [24,27], revealing that a high degree of parallelism is *available* in model evaluation, but that early vector and

	3	1	1	1
3	X	X	X	X
1	X	X	O	O
1	X	O	X	O
1	X	O	O	X

Figure 2.2(a) Non-optimal Ordering of Example Matrix

	1	1	1	3
1	X	O	O	X
1	O	X	O	X
1	O	O	X	X
3	X	X	X	X

Figure 2.2(b) Ordering of Example Matrix using Markowitz Criterion

array processors were not able to exploit this parallelism due to the gather-scatter bottleneck, as is detailed in Section 2.2.1.1. Further research on vector processors with special-purpose hardware for gather-scatter operation, described in Section 2.2.1.2, reveals that pre-processing of a sparse matrix can accelerate the sparse linear-equation solution process [34]. Special-purpose hardware approaches to parallel circuit simulation are presented in Section 2.2.2. Research with special-purpose hardware that efficiently exploits the high parallelism in the model-evaluation phase [23] is detailed in Section 2.2.2.1. In Section 2.2.2.2, a special-purpose hardware configuration designed for parallel linear-equation solution is described [56]. A hardware subcircuit-solver co-processor for use in a mixed relaxation-direct circuit-simulator multiprocessor system [17] is detailed in Section 2.2.2.3 and provides a hardware-based complement to the software approach to parallel processing taken in this dissertation. While the parallelism inherent in model evaluation has long been recognized [24], parallel sparse linear-equation solution has proven to be a more intractable problem. In Section 2.2.3, different approaches to parallel circuit simulation are described. An attempt at parallel, sparse linear-equation solution on a general-purpose multiprocessor is described in Section 2.2.3.1, serving to highlight some of the pitfalls to be avoided [56]. In Section 2.2.3.2, a parallel sparse linear-equation solution using dependence graphs proposed in [45] is described. A parallel version of the direct-method simulator, ADVICE [1] running on the Alliant FX-8 [40] is described in Section 2.2.3.3.

2.2.1 – Vector and Array Processors

Early vector processors, such as the CRAY-1 [26], were the first parallel computers available and used the Single-Instruction Multiple-Data (SIMD) stream mode of parallel computation. Thus, the pioneering work on developing parallel circuit simulators was

performed on vector and array processors and is described in this section. The first parallel circuit simulator, CLASSIE [24], was implemented on the CRAY-1 and is described in Section 2.2.1.1. CLASSIE performance indicated that the sparse irregular nature of the circuit matrix hindered efficient parallel simulation, thereby underlining the need for special-purpose hardware to handle the gathering and scattering of data between processing units and memory banks. The introduction of special-purpose hardware for gather-scatter operations resulted in significant improvement of circuit-simulation speeds on vector processors and one example of this improvement [34] is described in Section 2.2.1.2.

2.2.1.1 – CLASSIE: Circuit Simulation on the CRAY-1

A prototype circuit simulator, CLASSIE (implemented on the CRAY-1), is presented in [24] as a means of performing fast, accurate, hierarchical analysis of large-scale integrated circuits on vector computers. CLASSIE employs user-defined node tearing [57] to decompose a circuit into cells, according to functional (and structural) hierarchy, at the nonlinear-equation level. Identical cells are analyzed in parallel using vector operations.

In addition to vectorization, CLASSIE provides a number of modifications to standard circuit-simulation algorithms that improve its performance on a uniprocessor as well. Firstly, CLASSIE orders semiconductor devices by models, thereby saving time while gathering model parameters. Secondly, by decomposing a circuit into smaller subcircuits, CLASSIE circumvents the linear-equation solution-time problem of standard circuit simulators. This is because sparse linear-equation solution time in standard circuit simulators, which increases superlinearly with the number of nodes in a circuit, only dominates overall circuit-simulation time for large circuits (over 2000 nodes) [58]. It is believed that the

superlinear solution time is due primarily to longer searches through the sparse-matrix pointer structure (which would be aggravated by increased paging) and CLASSIE avoids this problem by reducing the dimension of individual sparse matrices. As in [19], CLASSIE employs code generation to reduce linear-equation solution time. Due to these modifications, CLASSIE runs twice as fast as SPICE2 on a VAX 11/780 uniprocessor for large circuits.

Although CLASSIE (on the CRAY-1) in its vector mode runs faster than it does in scalar mode and speed-up increases as circuit size grows, its performance is not encouraging on the whole. In particular, it is observed that data gather-scatter time for a bipolar junction transistor (BJT) takes as much as 75% of the total model-evaluation time. As a result, vectorized model evaluation in CLASSIE on the CRAY-1 yields a speed-up of about 1.5 for mixed BJT-diode circuits and about 2 for a MOS benchmark circuit. Vectorization results in a speed-up factor of under 2 for linear-equation solution, resulting in an overall simulation speed-up of less than an order of magnitude.

Thus, while the CLASSIE experiment indicates that substantial gains are available by decomposing and solving circuit-simulation problem data in parallel, the gather-scatter problem in vector processors such as the CRAY-1 makes realization of these gains difficult.

2.2.1.2 – Vector Circuit Simulation with Gather-Scatter Hardware

In [25], it is shown that matrix computation, the dominant part of large-scale simulation, is difficult to vectorize due to the sparse, irregular nature of the matrix. This corroborates the results obtained in [24] and described above in Section 2.2.1.1. However, it

has since been shown [34] that vectorized LU-decomposition can result in significant gains in simulation speed, *if the vector processor has special-purpose hardware for gather-scatter operations* (referred to as the "indirectly indexed vector feature"). Further, while CLASSIE processes structurally identical subcircuits in parallel, the vectorization algorithms used in [34] do not require user-defined subcircuits but automatically detect parallelism within the irregular structure of the matrix. The Block Vectorization Algorithm and Maximal Vectorization Algorithm used in [34] and their performance on the Hitachi S-810 [32] are described below.

The Block Vectorization Algorithm (BVA) for LU-decomposition identifies parallelism in blocks of consecutive columns of a matrix pre-ordered using the Markowitz criterion [54], such that the columns within a block can be decomposed independently of each other. Parallelism is exploited within each block using vectorized divide ($A_{ji} = A_{ji}/A_{ii}$) and update ($A_{jk} = A_{jk} - A_{ji} * A_{ik}$) operations. This algorithm allows the use of more efficient parallel dense-matrix operations towards the end of the LU-decomposition.

The Maximal Vectorization Algorithm (MVA) is derived from the BVA but attempts to locate parallelism *all over the matrix at every step of the LU-decomposition* and is not limited by the Markowitz ordering. The MVA computes a data reference level for each element, according to the earliest step at which the element can undergo division or updating. The LU-decomposition is then performed symbolically for the entire matrix and elements which produce the same data reference level are grouped together. Then, during the actual decomposition, all elements in the same group are processed in the same vector step.

In addition to the implementation of the BVA and MVA on the Hitachi S-810, the LU-decomposition is also accelerated through the use of code generation. For a 2132 MOS-transistor circuit, a factor of 16.5 speedup is achieved in matrix solution due to code generation alone. With the BVA and the MVA, overall speedups of 75.7 and 82.4 are achieved for matrix solution, indicating speedups of 4.59 and 5.0 due to the BVA and MVA respectively. While these results are encouraging and it is evident from [34] that the speedups increase with circuit size, the actual speed improvement due to exploitation of parallelism is still relatively small indicating the need for alternative algorithms. In addition, vector processors, being SIMD machines, require the static scheduling inherent in both the BVA and MVA and *force* all elemental operations to be of the same length. A general-purpose parallel processor, using an MIMD architecture, allows more flexibility in scheduling and can perform individual elemental operations according to their respective requirements, optimizing through the use of local memory.

2.2.2 – Special-purpose Hardware for Parallel Circuit Simulation

As a result of the increasing sophistication and reduced cost of VLSI, application-specific integrated circuits (ASIC's) are becoming feasible. Thus, it is now economical to synthesize hardware for the express purpose of performing a certain set of operations that would previously have been performed using software. To this end, numerous designs have been proposed that replace computation-intensive sections of circuit-simulation programs with special-purpose hardware. Descriptions of a scheme for parallel MOS model evaluation as well as one for large-scale parallel sparse linear-equation solution, both using attached processors, are presented below.

2.2.2.1 – MMAP: An Attached Processor for Parallel Model Evaluation

To exploit the high degree of parallelism inherent in the linearization or model-evaluation phase of circuit simulation [24,25,28,36] a special-purpose attached processor (MOS-Model Attached Processor, MMAP) that evaluates the dc-MOS transistor equations has been designed and evaluated in [23]. The special-purpose processor is attached to an IBM PC-XT personal computer, and the circuit simulator used is BIASC [59], a subset of SPICE written in the programming language C and designed to run on the IBM PC. A prototype MMAP utilizes pipelining and local memory to evaluate linear models for four MOSFET devices simultaneously. Experimental results indicate a high efficiency for parallel model-evaluation, limited by the 8-bit data bus of the IBM PC. The architecture of a system using the MMAP is shown in Figure 2.3.

The MMAP operates as follows: Transistor-model information is stored in the MMAP's local memory. When the host computer requires the evaluation of a transistor, it sends the transistor's input data, namely the model reference, terminal voltages and channel scale factor, to the MMAP, which evaluates the device coefficients at the operating point and returns the results to the host computer. Through the use of multiple-stage pipelined hardware, the MMAP is able to work simultaneously on several transistors. A novel empirical model, designed for use with the pipelined MMAP, enables model evaluation without conditional branching and uses only floating-point addition, subtraction and multiplication.

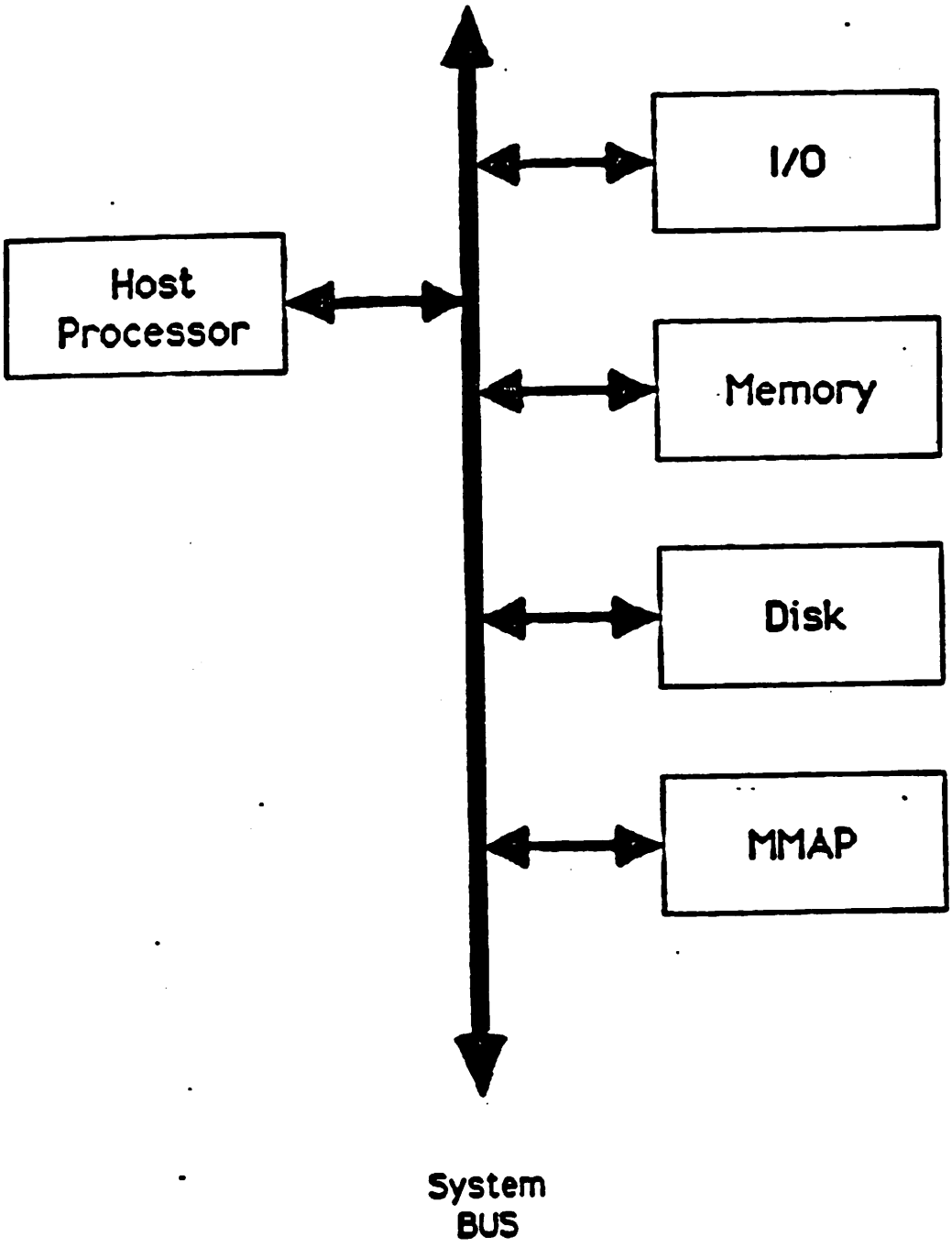

Figure 2.3 Architecture of the MOS-Model Attached Processor

While experiments with the prototype MMAP show that it incurs a 20% communication overhead, simulations of the MMAP indicate that with efficient operation of the system this overhead would be 60%, highlighting the slow transfer rate for floating-point data over the 8-bit data bus as well as the need for a wider and faster bus. In addition, it was observed that 40% of the MMAP's active time was spent transferring data between the Model Processing Unit (MPU) and the MMAP's memory, which were on separate chips, indicating that a single-chip implementation of the MPU with coefficient memory would significantly enhance attached-processor efficiency.

2.2.2.2 – BLOSSOM: A Systolic Array for Parallel Linear-Equation Solution

A comprehensive study of various techniques for the solution of large-scale linear systems of equations has been conducted in [56]. This study includes analyses of the best uniprocessor algorithms, as well as special-purpose hardware and multiprocessor algorithms obtained by mapping the best uniprocessor algorithms on to different architectures. The various techniques are compared on the basis of numerical stability, sparsity preservation, computational complexity and convergence rates, while recognizing the dependence of parallel-algorithm performance on the parallel-processing environment, which is difficult to model accurately.

For large-scale systems, in the interests of preserving sparsity and maintaining low computational complexity, Gaussian elimination with pivoting is identified as the most efficient means of solving sparse linear equations. Further, block LU-factorization, using inverses of the diagonal submatrices rather than of their L and U factors, is selected as the basis for the proposed special-purpose hardware for parallel linear-equation solution.

BLOSSOM, the special-purpose architectural system proposed for parallel, sparse linear-equation solution, comprises of a reconfigurable systolic array connected to a host computer. BLOSSOM uses its own memory, data bus and executive control unit to operate independently once the system to be solved has been loaded into its local memory by the host. Submatrix operations to be performed by the processors in the systolic array are assumed to be implemented as microprograms in the processors. The system architecture with the BLOSSOM unit is shown in Figure 2.4.

Results from a software simulator of the BLOSSOM system indicate that most of the partitions are 2×2 submatrices; for a 1957×1957 matrix (from the simulation of a memory

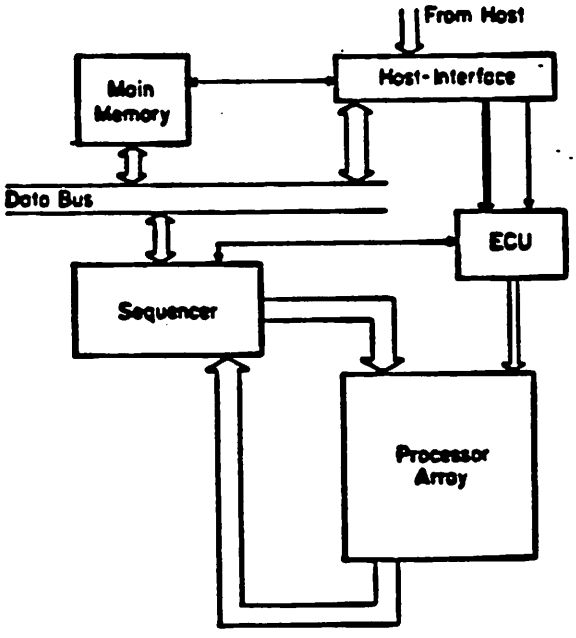

Figure 2.4 Architecture of the BLOSSOM Linear-Equation Solution System

circuit) it is seen that a 2×25 processor array solves the system 9.25 times faster than a 2×2 array, indicating 74% efficiency. Given the high efficiency of the BLOSSOM system, even without exploiting the parallelism that would be available by pipelining the submatrix operations, it is evident that such a hardware-based approach shows promise for rapid parallel sparse linear-equation solution.

The BLOSSOM experiment displays potential for accelerating sparse linear-equation solution through the use of costly special-purpose hardware. A more inexpensive alternative is the use of general-purpose multiprocessors, as is described in Section 2.2.3.

2.2.2.3 – A Special-Purpose Subcircuit Solver

As mentioned in Chapter 1, there are circuit simulators, such as SPLICE and RELAX, that utilize a mix of direct and relaxation methods to solve the circuit equations. While subcircuits in a direct-relaxation scheme can be solved concurrently using relaxation [60], each of the subcircuit solutions itself is a direct-method solution and is not suited as well to parallel processing. Thus, a specialized subcircuit solver to perform the direct-method solution in parallel has been designed [17] and is discussed briefly below.

The special-purpose subcircuit solver is a five-way parallel processor operating in pipelined mode on independent instruction streams, with 512k bytes of local memory and a 128-bit datapath to memory. The pipelined implementation here, as in MMAP, circumvents the need for branching due to dependencies. The 512k bytes of memory are shared between the 5 processors as well as the host processor, eliminating the need to transfer data after a subcircuit solution, while the wide datapath is designed to accommodate a (64-bit) double-precision floating-point number in conjunction with a 32-bit pointer and a 32-

bit integer. This permits simultaneous processing on all three fields of the data, allowing the co-processor to step through a linked list while performing operations on matrix elements, which is useful when working with sparse matrices.

Although no results are published in [17], it is evident that the subcircuit solver exploits parallelism primarily during the model-evaluation phase. This is appropriate since subcircuits typically contain a small number of circuit nodes (less than 100) hence model evaluation dominates overall simulation time. However, this approach will not work efficiently for large circuits since then sparse linear-equation solution dominates overall simulation time.

2.2.3 – Multiprocessor Sparse Linear-Equation Solution

As stated early in Chapter 1, parallel processing is attractive as a technique to reduce circuit-simulation runtimes for large VLSI circuits. As shown in Section 2.2.2, a number of researchers have identified and successfully exploited the parallelism available in the model-evaluation phase of circuit simulation, but sparse linear-equation solution, which dominates simulation time for large circuits, continues to be difficult to parallelize. In this section, two techniques that utilize general-purpose multiprocessors to exploit the parallelism in sparse linear-equation solution are described.

2.2.3.1 – Parallel Linear-Equation Solution: Dynamic Pivot Ordering

On multiprocessor systems, it has been shown that Gaussian elimination can be efficiently parallelized when solving dense, large-scale linear systems [61] and that nested dissection techniques allow efficient parallel solution of well-ordered sparse linear sys-

tems [11]. However, circuit matrices, in addition to their extreme sparsity (three to four non-zero elements per row or column), have an *irregular* nature that makes most decomposition techniques, including nested dissection, difficult to use [57].

In [56] a parallel algorithm based on LU-decomposition, called *Segmented Partial Pivoting (SPP)*, is implemented on the BBN Butterfly Parallel Processor as a first step towards parallelizing the solution of irregular, sparse linear systems. In the SPP algorithm, each processor is assigned a block of rows from the system matrix; for each column in the matrix, all the processors work within their assigned blocks to determine local pivot candidates and then to eliminate other non-zero elements in the column. Next, the processors compete to set a lock. The local pivot of the processor that sets the lock becomes the *global pivot* for the given column, and the other processors use the global pivot to eliminate their own local pivots. This process is repeated until the system matrix is triangular. Forward elimination is also performed in parallel, as is back substitution. These operations employ parallel multiplication (by matrix coefficients) and subtraction (from the right hand side) after an unknown has been solved for, as well as parallel solution (division) which is made possible by the sparsity of the matrix.

Experimental results using the SPP algorithm on the BBN Butterfly indicate a high speed-up using up to about ten processors, but no improvement beyond that. Further, the competitive global pivot-selection scheme results in a large number of fill-ins, indicating that even with the parallelism exploited parallel solution may not be faster than efficient uniprocessor linear-equation solution. This study thus leads to two conclusions: first, there is a relatively low degree of parallelism available in the parallel linear-equation solution for circuit matrices; secondly, schemes that attempt to exploit as much of the small parallelism

inherent in this solution process may result in excessive fill-ins, thereby nullifying their advantage.

2.2.3.2 – Linear-Equation Solution: Static Dependence Graphs

As shown in Section 2.2.3.1, significant cost is associated with allowing fill-ins during LU-decomposition for circuit matrices. Thus, even for parallel solutions, a scheme that attempts to minimize the number of fill-ins, such as the Markowitz criterion, is desirable. In [45] a task system represented by an acyclic directed graph is statically built *after* the matrix has been ordered to minimize fill-ins. The task graph is assembled representing *divide* and *update* operations as nodes, and inter-element dependences as arcs.

For a 400×400 microwave-amplifier sparse matrix of density 0.8%, simulations indicate that 100 processors could be kept busy 73% of the time for matrix triangulation and 33.8% of the time for back-substitution. The results also indicated that solution time decreases inversely with the number of processors for small numbers of processors (10-20% of the order of the matrix); however, as the number of processors increases the amount of parallelism in the matrix is unable to keep the processors busy and the time for triangulation is limited by D, the depth of the task graph (the number of tasks along the critical path of the graph). These results are noteworthy because they demonstrate that a significant amount of parallelism is available in sparse linear-equation solution using standard sparsity-directed ordering of the matrix. The Pivot Dependency Graph (PDG) scheme used in this dissertation, described in Chapters 4 and 5, is similar to the task graph scheme and provides insight into practical issues associated with implementing dependence-graph algorithms for sparse linear-equation solution.

2.3: Conclusions

While different algorithmic techniques have proved more efficient than standard direct methods for the simulation of certain classes of circuits, direct-method simulators continue to be important. In the first section of this chapter a brief overview of direct-method simulation algorithms is provided. A number of hardware-based acceleration techniques, described in Section 2.2, have been studied towards reducing runtimes for direct-method simulation. All of these techniques, however, have been restricted to either *special-purpose hardware* or single phases of the direct-method algorithm. In this dissertation, parallel algorithms for direct-method circuit simulation are studied using general-purpose multiprocessors.

CHAPTER 3

PARALLELISM LIMITS IN CIRCUIT SIMULATION

Prior to parallelizing a particular technique for solving a problem, it is helpful to establish theoretical limits to the basic parallelism available in the problem. Theoretical limits can be determined by studying the realizable parallelism in the most-parallel algorithm known for the solution of the problem. If the theoretical study shows the most-parallel algorithm has poor potential for parallelism, all parallel algorithms are bound to perform poorly. Conversely, high theoretical parallelism limits, although not an assurance of good performance, indicate that further research into other algorithms may not be fruitless.

To determine a limit to parallelism in circuit simulation, a node-based relaxation circuit-simulation algorithm, MSPLICE, has been implemented [62, 63, 64] on the BBN Butterfly Parallel Processor [37]. Various scheduling and shared-memory storage schemes have been investigated using both the initial implementation of MSPLICE (MSPLICE1) as well as an improved implementation (MSPLICE2 [65]). Listings of MSPLICE1 and MSPLICE2 are provided in Appendices B and C respectively. While the initial implementation of MSPLICE described in [62] was developed for an early version of the Chrysalis Operating System [66], MSPLICE1, although using the same algorithms, is a modified version of MSPLICE designed to run on the Chrysalis 2.2 version of the Butterfly operating system [67]. Analysis of empirical data from the MSPLICE experiment described in this chapter verifies that circuit simulation is well-suited to parallel processing at the node-based (or fine-grained) level. This fact justifies further research into parallelizing general-purpose circuit simulators.

MSPLICE1 has been implemented on both the Digital VAX 11-780 and the BBN Butterfly machine. Only a 10-processor implementation of the Butterfly was available for initial MSPLICE1 development [62]. Further results using MSPLICE1 running on up to 78 processors are presented here. With such a large number of processors, additional insights into multiprocessor-based circuit simulation have been obtained and significant bottlenecks in MSPLICE1 have been removed in the new implementation, MSPLICE2. Further, a version of the Butterfly multiprocessor has been developed which supports up to 256 processors [67]. Consequently, the upgraded program, MSPLICE2, has the capability of utilizing up to 256 processors in parallel. MSPLICE2 results presented in this chapter are for the 101 processors of a 128-processor Butterfly system that were functional at the time benchmark studies were performed. In addition to the Ideal Gauss-Seidel Machine presented in [62], a new technique has been developed [68] to determine the maximum amount of latency available in the test circuits. Together, these meters provide a meaningful upper bound for measuring the parallel performance of MSPLICE with various benchmark circuits.

In Section 3.1, the motivation for selecting the MSPLICE algorithm to measure parallelism limits is provided. The Iterated Timing Analysis (ITA) and MSPLICE algorithms are reviewed in Section 3.2, while the Butterfly architecture and programming environment are described in Section 3.3 and the benchmark circuits used in this study are introduced in Section 3.4. An analysis of the parallel performance of MSPLICE1 on up to 78 processors is presented in Section 3.5. Based on MSPLICE1 performance, algorithmic modifications were made to create the program MSPLICE2, which circumvents bottlenecks identified in the original implementation. The modifications and results from the new implementation are described in Section 3.6, and the conclusions observed through the MSPLICE

experiment are summarized in Section 3.7.

3.1: Motivation for using MSPLICE as a measurement tool

For any parallel algorithm, there are a number of factors that determine the efficiency with which the algorithm will perform. The four most important factors are listed below:

(1) The *amount of parallelism* in the algorithm, i.e., the number of tasks that can be executed independently at any stage of the computation,

(2) The ratio of task *computation time* to intertask *communication and synchronization time*,

(3) The *granularity* of the tasks, i.e. the size of the smallest tasks, and

(4) *Load balancing*, or distribution of tasks between processors.

The selection of a node-based relaxation simulator has been made in light of these four factors and is described below.

The MSPLICE algorithm is well-suited to parallelization primarily because it uses relaxation which intrinsically provides a high degree of parallelism. Since a relaxation simulator decouples the circuit equations, it solves them *independently* of each other, iterating to ensure convergence to the correct solution. Thus, relaxation simulators provide an implicit partitioning of a problem into a number of independent tasks. This is important for parallel processing since each task can be executed on a separate processor, with no inter-processor communication or synchronization. Inter-processor communication degrades parallelization as it requires processors to synchronize with each other, thereby

inhibiting uninterrupted program execution.

In addition to being node-based, the MSPLICE algorithm was also selected as a test-bed to measure limits to parallel circuit simulation due to its node-based nature. A node-based simulator intrinsically partitions a circuit into small and fairly uniform tasks, which results in efficient load balancing. *Load balancing*, the equal distribution of work across a number of processors, is best illustrated by the following analogy: given the job of placing either grains of sand or (larger) stones in a number of jars, the grains of sand provide for a more equitable weight distribution between the jars. Efficient load balancing is important for parallel processing as it ensures that all the processors finish their work at approximately the same time, thereby reducing time wasted at synchronization barriers.

Although it is node-based and uses relaxation, both qualities desirable for parallel processing, MSPLICE has not yet proven to be a generally applicable circuit-simulation algorithm. MSPLICE is used only to determine the *potential for parallelism* in circuit simulation, with the recognition that it may not always be efficient as a simulation algorithm.

3.2: Algorithms used in MSPLICE

MSPLICE is a parallel algorithm based on the *Iterated Timing Analysis* (ITA) [7] relaxation-based approach. The ITA method, described briefly in Section 3.2.1, is a successive-over-relaxation Newton method which uses event-driven analysis and selective-trace to exploit the temporal sparsity of the electrical network [7]. Because event-driven, selective trace techniques are employed, the algorithm lends itself to implementation on a data-driven computer. The MSPLICE worker algorithm that runs on each processor is

described below in Section 3.2.2.

3.2.1 – Iterated Timing Analysis (ITA)

The ITA method is a form of electrical analysis derived from timing simulation [7,69]. The starting point for a description of ITA is the electrical circuit equation formulation. A Nodal Analysis [47] formulation will be used to illustrate the ITA algorithms. The following assumptions are used:

- All resistive elements, including active devices, are characterized by constitutive equations where voltages are the controlling variables and currents are the controlled variables.
- All energy storage elements are two-terminal, possibly nonlinear, voltage-controlled capacitors.
- 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.

In the nodal network equations there are N equations in N unknown node voltages. For the circuit $N+1$ nodes are present, where node $N+1$ is the reference node, or ground. The equations can be written as:

$$C(v,u)\dot{v} = -f(v,u) \qquad (2.1)$$

$$v(0) = V.$$

where $v(t) \in \mathbb{R}^n$ is the vector of node voltages at time t, $\dot{v}(t) \in \mathbb{R}^n$ is the vector of time derivatives of $v(t)$, $u(t) \in \mathbb{R}^n$ is the input vector at time t, $C(\bullet) : \mathbb{R}^n \rightarrow \mathbb{R}^{n \times n}$ represents the nodal capacitance matrix, $f : \mathbb{R}^n \times \mathbb{R}^n \rightarrow \mathbb{R}^n$, and:

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

where $f_i(v(t), u(t))$ is the sum of the currents charging the capacitors connected to node i. The differential equations are converted to a set of nonlinear, algebraic difference equations using a stiffly stable integration formula to give:

$$g(x) = 0 \tag{2.2}$$

where $x \in \mathbb{R}^N$ is the vector of node voltages at time t_{n+1} and $g()$ is the non-linear function. An iterative method, e.g., Gauss-Jacobi or Gauss-Seidel relaxation, is used to solve the equations. However, unlike timing analysis where a single relaxation iteration is used per time-point, in the ITA approach *the relaxation process is continued to convergence* at a time-point.

Only one Newton-Raphson iteration is used to approximate the solution of each nodal equation per relaxation iteration. Event-driven selective-trace techniques may still be used to exploit latency, as for timing simulation. Since in ITA the nonlinear circuit equations are solved by an iterative method until satisfactory convergence is achieved, the numerical properties of the integration methods used to discretize the circuit equations are retained. Thus, the stability and the accuracy problems typical of the timing simulation algorithms are not an issue here [42].

The following algorithm, written in "Pidgin 'C'" [70], illustrates the principle steps involved in ITA analysis, using a Gauss-Seidel iteration, for use on a conventional computer. At each time at which one or more nodes are scheduled to be processed, two event lists, $E_k(t_n)$ and $E_{k+1}(t_n)$ are used to separate the nodes to be processed in successive iterations, k and $k+1$, of the Gauss-Seidel-Newton process.

Gauss-Seidel-Newton Iteration:

Put all nodes that are connected to independent sources in event list $E_k(0)$;
$t_n = 0$;
while ($t_n < TSTOP$) {
  $k \leftarrow 0$;
  while (event list $E_k(t_n)$ is not empty) {
    foreach (i **in** $E_k(t_n)$) {

$$v_i^{k+1} = v_i^k - \frac{g_i(\bar{v}^{k+1,i})}{g_i'(\bar{v}^{k+1,i})};$$

      where $\bar{v}^{k+1,i} = [v_1^{k+1}, \cdots, v_i^{k+1}, v_{i+1}^k, \cdots v_N^k]^T$
      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 $E_k(t_s)$;
      } **else** {
        add node i to event list $E_{k+1}(t_n)$;
        add fanout nodes of node i to event list $E_k(t_n)$ if not already on $E_k(t_n)$;
      }
    }
    $E_k(t_n) \leftarrow E_{k+1}(t_n)$;
    $E_{k+1}(t_n) \leftarrow$ 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 in the time queue at which an event was scheduled. In this way, the "time-step" is handled independently for each node. The **foreach** construct requires that the block be executed for each member of the set in a specified order.

This simplified algorithm does not illustrate how such issues as time-step reduction and local truncation-error estimation are handled. These and other important details of the algorithm are described elsewhere [7]. While a nodal formulation was used to describe the approach, a Modified Nodal formulation [49] can also be derived.

3.2.2 – The MSPLICE Algorithm

MSPLICE uses the data partitioning approach to multiprocessing, where a number of processors perform identical functions on separate parts of the problem data. Nodes described using Nodal Analysis methods are statically stored in the memories of different processors. Typically, the number of nodes in the circuit greatly exceeds the number of processors in the multiprocessor system. Thus, a number of circuit nodes are allocated to each processor's memory. Having allocated data pertaining to particular nodes in the different processors, one can choose either of two approaches to task scheduling. A task is defined as the evaluation of a particular node voltage and is described below. In MSPLICE1, task scheduling can be done either statically, where a processor is responsible for evaluating voltages of all nodes that reside in its memory, or dynamically, where a processor is assigned a task when it becomes free. In the event of dynamic scheduling, which aids load balancing, a central scheduler can be used to distribute tasks to processors. This produces a serialization bottleneck. Instead, both the MSPLICE implementations, as described in Sections 3.5 and 3.6, use distributed schedulers, where each processor is responsible for its own scheduling.

In MSPLICE, a single global variable called *GlobalRemainingNets* is used to coordinate the processors at a given time point. It is incremented whenever a node is scheduled at a particular time point and is decremented when a node has finished being processed. When *GlobalRemainingNets* reaches zero, all processors move to the next time point of the simulation.

From the point of view of a single processor, P, once it has been allocated a set of electrical nodes, M, it proceeds as follows at time t_n: -

```
foreach ( node i in M scheduled at t_n ) {
/* STEP (1): */
   foreach ( fanin element at i )
      obtain its fanin node voltages, v_j^K,  j≠i, K = k or k+1;
/* STEP (2): */
   foreach ( fanin element at i )
      compute its contributions to nodal equation;
   obtain v_i^{k+1} using a single Newton-Raphson step as described in Chapter 2;
   if ( convergence is achieved ) {
      if ( v_{i,n} ≠ v_{i,n-1} ) {
         schedule i at t_{n+1};
         decrement GlobalRemainingNets;
      }
   }
   else {
      schedule i again at t_n;
      forall ( fanout nodes of i ) {
         increment GlobalRemainingNets;
         Schedule fanout node at t_n according to scheduling convention.
      }
   }
}
```

3.2.3 – The Ideal Gauss-Seidel Machine

For any circuit simulated using the MSPLICE algorithm described above, there is an *ideal* performance that would be expected of an ideal multiprocessor that incurs no scheduling or communication overhead. This performance is predicted by an Ideal Gauss-Seidel Machine (IGSM) [62, 64, 63]. The Ideal Gauss-Seidel Machine simulates an ideal multiprocessor for the Gauss-Seidel algorithm, where communication time between processors is negligible with respect to local computation time and all communication is non-blocking. In addition, each processor in the IGSM can solve an equation (i.e., process one task) in a single unit of time.

The IGSM model ignores such factors as contention for shared global data and the overhead of queue handling, which degrade performance of a parallel program as the

number of processors increases. Further, in its assumption that each processor takes a single unit of time to solve a single node equation, the IGSM implicitly assumes that all nodes have the same number and type of devices in identical states.

Although the IGSM model is an optimistic one, it does provide a theoretical limit to the amount of parallelism available during the simulation of a particular circuit. The degree of optimism in the IGSM model becomes more apparent as the applications upon which it is based become less valid. Actual simulation times are used, in Section 3.6.3, to estimate the accuracy and the limitations of the predictions from the model of the ideal machine.

3.3: The BBN Butterfly Parallel Processor

MSPLICE has been implemented on the BBN Butterfly (TM) Parallel Processor[37] (see Figure 3.1), which is described in this section. The Butterfly has been chosen as a test-bed multiprocessor for its high-speed Omega interconnection network [71]. The high-speed network results in a ratio of remote-memory access time (4 microseconds) to local-memory access time (625 nanoseconds) near 6, which is similar to the uniprocessor ratio between a cache miss and a hit [37]. In addition, the Omega network is extensible to 256 processors, making the Butterfly useful for the investigation of the performance of an algorithm as the number of processors increases.

The Butterfly comprises a number of identical processor-memory elements linked to each other through the Omega-connected network, which is built using high-performance switches (see Figure 3.2). The combination of the local memories of all the processors in the system acts as a single, shared global memory, with each processor able to make refer-

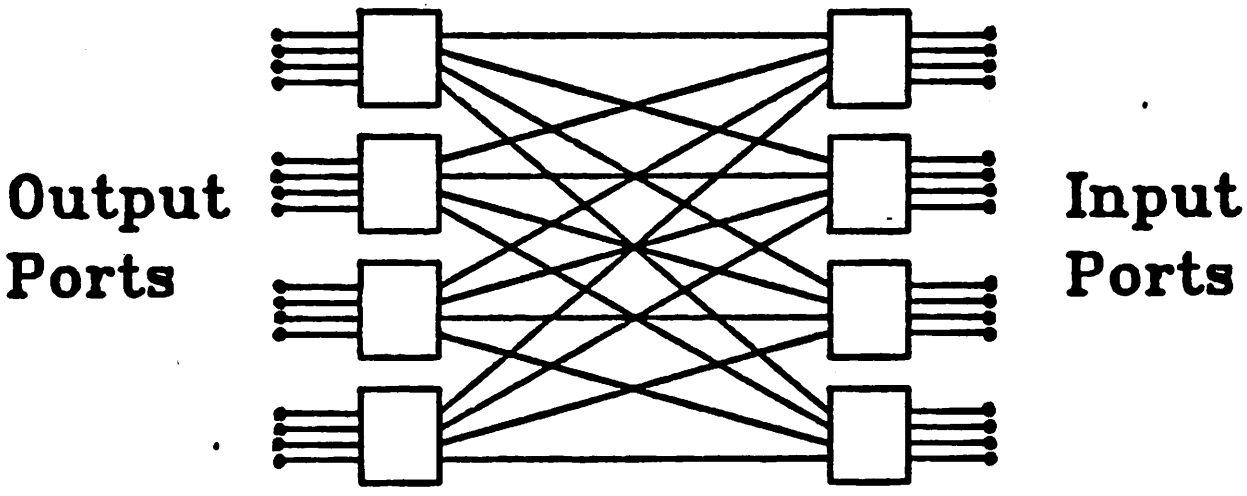

Figure 3.1. The Architecture of the BBN Butterfly Parallel Processor

ences to memories other than its own. Thus, once a datum is placed in shared memory, the responsibility for determining where the datum is located falls on the individual processors, rather than on the programmer. This makes programming easier, although it may affect program efficiency. For the Butterfly, the fast interconnection-network usually overcomes the degradation in performance due to remote accesses.

3.3.1 – The Butterfly Architecture

In addition to a processor and memory, each independent module of the Butterfly, referred to as a Processor Node, also comprises a Processor Node Controller (PNC), an input-output (I/O) bus, an interface to the Butterfly switch and a start-up EPROM, shown

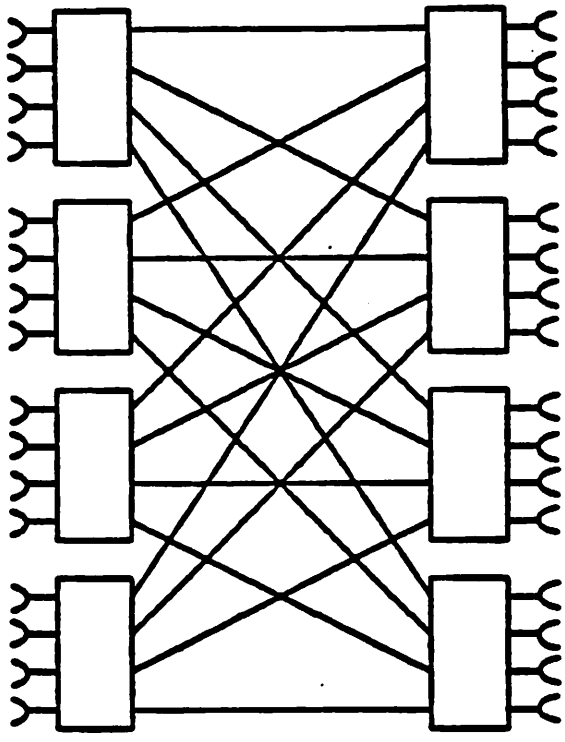

Figure 3.2. 16-processor Butterfly Switch

in Figure 3.3. The I/O bus is responsible for interaction between the Processor Node and the host machine, as in the reading or writing of data from or to the host's disk. The processor performs computations on data resident in either its own (local) memory or the (remote) memory of other processors. The PNC determines whether a datum addressed by the processor is in local or remote memory and then accesses the datum through the Butterfly switch. Each of the three major components of the Processor Node, i.e. the processor itself, the PNC and the Butterfly switch, is described in more detail below.

The Processor Nodes in the Butterfly Parallel Processor for the experiments described in this chapter used the Motorola MC68000 microprocessor. More recently, the MC68000

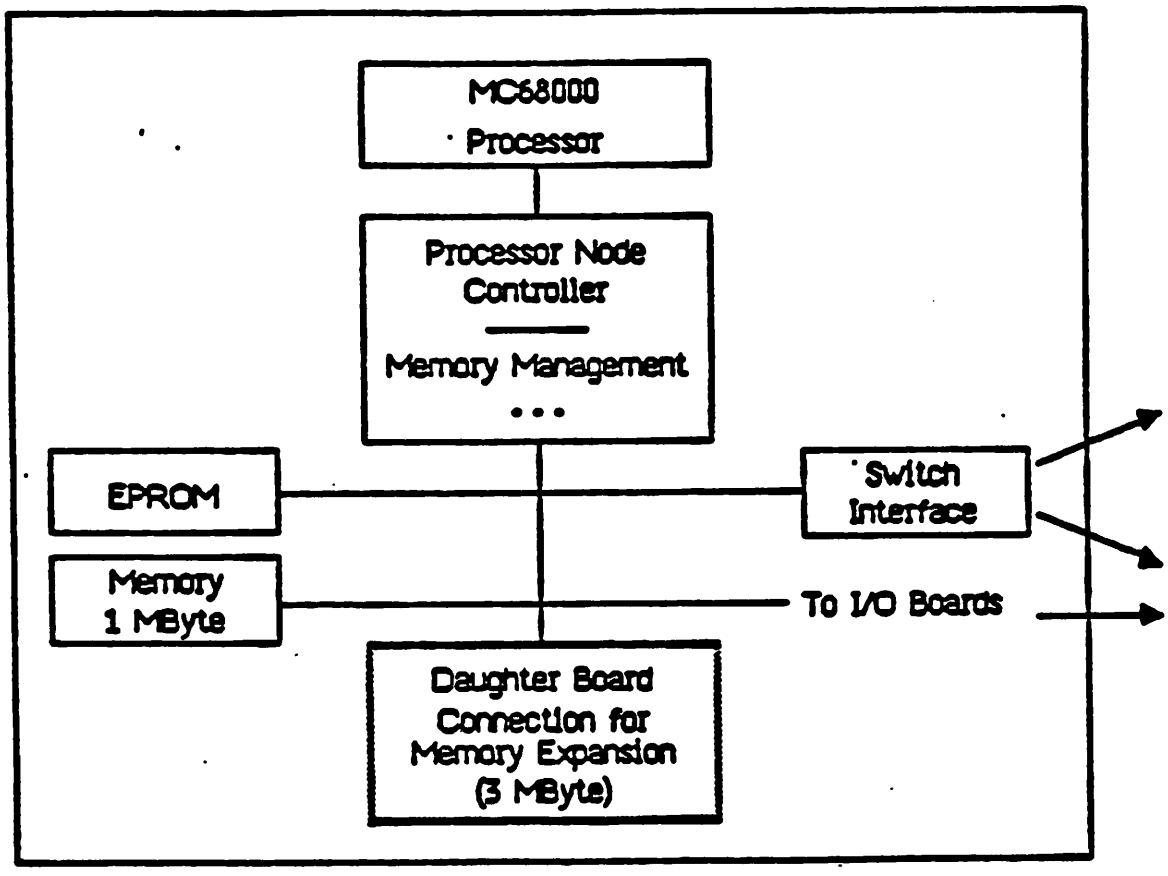

Figure 3.3. Block Diagram of a Butterfly Processor Node

has been replaced by a platform that holds an MC68020 microprocessor and a M68881 floating-point unit, which significantly enhances Butterfly performance, especially for floating-point intensive applications like circuit simulation. The effect of this enhancement is to reduce a single task computation time, thereby decreasing the ratio of computation to communication time for each task. Consequently, as is shown at the end of Section 3.6, speedup performance of MSPLICE2 is degraded due to the improved technology. Normally, however, new technology would improve both the processor as well as the interconnection network, resulting in no relative change between computation and

communication time.

The PNC intercepts all the associated processor's memory references and determines where the addressed data is located. If the data is resident in local memory, it is accessed directly; if, however, the data is in another processor's memory, the PNC sends a request message to the other processor's PNC through the Butterfly switch. When the remote PNC receives the message, it accesses the requested datum and sends a reply message to the first PNC, which, if necessary, communicates the result of the transaction to its associated processor. The PNC also provides efficient microcoded implementations of a number of atomic operations, such as queue handling, test-and-set and scheduling. As a result, it contributes significantly to efficient parallel performance of the Butterfly.

The Butterfly switch uses packet-switching techniques between sets of switching nodes configured as a "serial-decision" network. The topology used is similar to that of the Fast Fourier Transform Butterfly algorithm [72], hence the name. Each switching node has four inputs and four outputs, and the entire configuration ensures that there is a path through the switch network from each processor node to every other node in the system. Communication between the processors is performed via message "packets", that contain the address of the receiving node and the data to be transferred. At each step of the switching network, part of the address is stripped and the remaining message is transmitted according to the rest of the address, as illustrated in Figure 3.4. The switch has a bandwidth of 32 Mbit/sec, which is needed when performing block transfers. In the event of contention for a particular switching node output port, one of the messages is allowed to proceed, while the other is re-transmitted after a short delay.

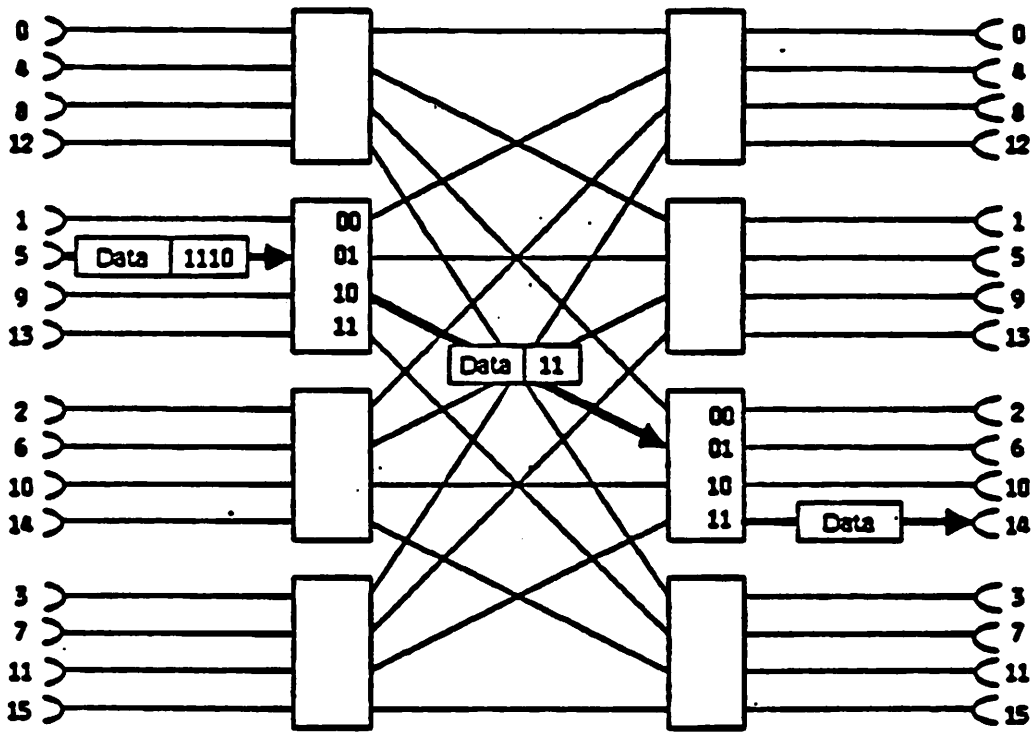

Figure 3.4. A packet in transit through the Butterfly switch

3.3.2 – The Butterfly Programming Environment

Applications on the Butterfly operate under the Chrysalis operating system [67]. MSPLICE1 and MSPLICE2 are programmed in the C programming language, although the Butterfly supports both Fortran and Lisp as well. Programs for the Butterfly are written, modified and cross-compiled on the host front-end system, and then loaded and run on the Butterfly. The Chrysalis operating system [67] provides a rudimentary Unix-like high-level system, as well as a library of low-level subroutines. Of prime importance for paral-

lel processing are the process control, memory management and synchronization primitives provided. Process schedulers use real-time, flexible scheduling algorithms that are microcoded for fast process switching. A segmented, virtual-memory management system provides access to up to 4 Gigabytes of memory, although only 16 Megabytes can be addressed at a time. To address memory outside the current 16 Mbytes, a new set of Segment Attribute Registers must be mapped in, which is a time-consuming process on the present implementation. As regards interprocess communication, the data structures provided include dual queues and events, which are handled in the microcode for fast interprocess communication.

Chrysalis treats all data structures on the Butterfly as *objects*. Each object is addressed by a 32-bit *object handle*, which specifies the 8-bit address of a processor and a 24-bit virtual address. When a memory object is to be used by a process, the object is *mapped* into the address space of the process. At this time, the memory object, or segment, is assigned a new Segment Attribute Register (SAR) in the process' SAR array. Subsequently, the processor node is capable of addressing any data within the memory segment, regardless of the segments actual physical location.

Events and *dual queues* are two synchronization mechanisms provided by Chrysalis that serve to make parallel programming on the Butterfly easy and efficient. Events are created by server processes and sent to client processes. When clients require service from the server process, they *post* the event to the server process. The server process may or may not react to the event posted and may have a number of outstanding events at any time. Events are identified by *event-handles*, which are recognized across the entire system. Dual queues provide an efficient queuing scheme, by utilizing events to prevent

busy-waiting on a queue. A dual queue can hold either data or event-handles, but not both concurrently. When a server attempts to dequeue a datum from the dual queue, if the queue is empty or only contains event-handles, the server will place its own event-handle on the dual queue. Subsequently, when a datum is placed on the dual queue, the server process whose event-handle is at the top of the queue is notified by the client process enqueing the datum. Thus, instead of servers constantly polling the queue for work, they are notified of work as and when it arrives.

In addition to normal programming considerations, parallel programming also requires efficient, easy processor and storage management. On machines with large interprocess communication time, the most efficient paradigm for process and data management is that of *cooperating sequential processes*, where each process operates on independent data, occasionally sending messages to other processes in the system [73]. However, there are numerous applications where large, independent tasks are difficult to identify; hence, parallelism must be exploited at a far finer granularity. For such applications, a *shared-memory environment* is far more desirable, where all the processors can simultaneously work on the same data, without having to alert each other of modifications explicitly. The Uniform System, a programming environment that uses Chrysalis primitives to implement a shared-memory system, has been implemented on the Butterfly. In order to keep all memories equally busy, the Uniform System provides a set of routines that allow the user to distribute shared memory across the entire processing system. As a result, contention for shared memory is distributed across the system, and hot spots are minimized. The Uniform System treats processors as a group of identical workers, each able to perform any task. Thus, an application must be broken into two functionally separate parts: *generators*, which identify the next task to be performed, and *workers*, which actually perform the task itself.

3.4: Benchmark Circuits

A number of benchmark circuits have been used for this analysis and for the subsequent identification of performance bottlenecks. The first of these circuits is a digital-filter circuit (Digfi) whose netlist and parasitic capacitor values were obtained from an analysis of the mask layout of an industrially designed circuit. Other benchmark circuits used for the analysis of MSPLICE1's performance and early MSPLICE2 performance include a chain of 50 NMOS inverters (Chain) and a four-bit binary counter (Counter).

An *a posteriori* analysis of all of the waveforms generated by MSPLICE has been used to determine the maximum amount of latency available in the three benchmark circuits used [68]. The procedure used is as follows: two sets of waveforms for all circuit nodes are simulated, using MSPLICE and a direct-method circuit simulator. The number of timepoints, M, simulated using MSPLICE are compared against the corresponding number of timepoints, D, used by the direct-method simulator. The percentage ratio

$$\frac{(D - M)}{D} \times 100\%$$

provides an indication of the amount of latency *actually* exploited by MSPLICE. The *a posteriori* analysis of the waveforms produced by the direct-method simulator is used to estimate how many points *ideally* needed to be used. The resulting number of timepoints, I, provides a measure of the activity in the circuit. The percentage ratio

$$\frac{(D - I)}{D} \times 100\%$$

indicates the maximum latency that can be exploited during the simulation of the circuit. Actual (using MSPLICE) and ideal latency exploitation figures for the benchmark circuits are presented in Table 3.1, along with dimensional information about the circuits. The small exploitation of latency for the inverter-chain circuit causes more of the circuit to be

active at each time point. This results in a larger number of tasks and this circuit is thus capable of keeping a large number of processors busy, as is shown in later sections.

Ckt.	# of Nodes	# of MOSFET's	Actual Latency	Ideal Latency
Chain	54	102	12.5	54.5
Counter	222	517	39.0	49.0
Digfi	385	698	70.0	78.0

Table 3.1: Characteristics of the Benchmark Circuits

All three circuits have been simulated on the ideal Gauss-Seidel machine model [62, 63, 64], as shown in Table 3.2 below.

#Procs	Chain	Counter	Digfi
1	1.00	1.00	1.00
2	1.98	1.98	1.97
4	3.85	3.90	3.81
8	7.26	7.51	7.28
16	12.94	14.00	12.82
32	21.34	24.32	20.66
64	32.05	38.35	28.82
128	38.33	53.52	34.87

Table 3.2: MSPLICE Performance Predicted by the Ideal Gauss-Seidel Machine

The model predicts that the marginal utilization falls as additional processors are added since there is a maximum limit to the parallelism available in the circuit at any timepoint. Thus the ideal machine model provides an upper-bound for the performance of a real machine simulating the circuit and using the ITA algorithm.

3.5: The MSPLICE1 Program

The MSPLICE1 program, developed by Deutsch [62], was the first implementation of the MSPLICE algorithm and was run on 10 processors of the BBN Butterfly. On 10 processors, both static and dynamic scheduling schemes performed equally well. In this section, the performance of the MSPLICE1 program on up to 78 processors of the Butterfly is described and indicates shortcomings in both schemes, thereby providing the direction for the next generation of the MSPLICE simulator.

3.5.1 – MSPLICE1 Performance: Multiple-Queue Dynamic Scheduling

Figure 3.5 shows the results obtained while simulating the benchmark circuits on up to 78 processors, using dynamic allocation of tasks to processors. It is evident that with up to about 40 processors, the experimental results compare reasonably with the predicted performance of the ideal Gauss-Seidel machine. With 32 processors, the digital-filter circuit simulation runs 68% as fast as predicted by the ideal. However, as the number of processors is increased further, performance falls off relative to the ideal Gauss-Seidel model. In addition, after a certain point the performance begins to fall off in absolute terms as well, implying that the additional processors are degrading rather than improving performance. This runs contrary to the predictions of the ideal Gauss-Seidel machine which claims

#Procs	Chain	Counter	Digfi
1	1.00	1.00	1.00
2	1.83	1.84	1.82
4	3.15	3.48	3.14
8	6.15	6.60	5.76
16	9.56	11.21	9.62
32	9.67	14.93	12.00
64	7.94	11.02	10.62
78	7.52	8.73	9.81

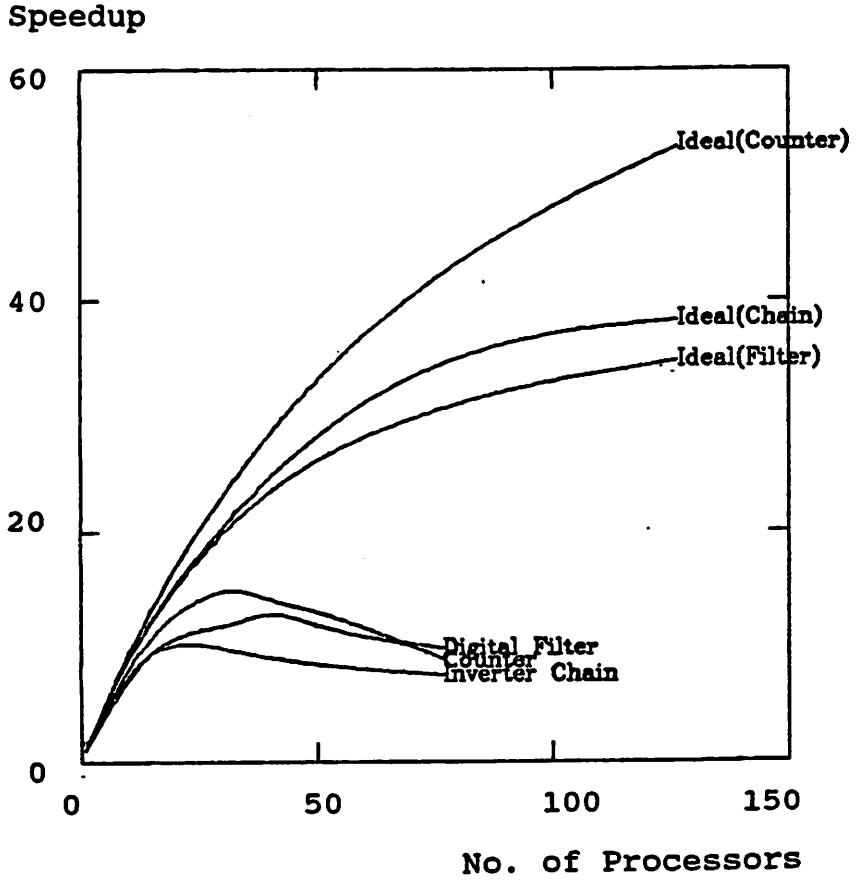

Figure 3.5. Performance curves for MSPLICE1 using distributed, dynamic task allocation.

uniformly increasing improvement with up to 128 processors.

The above experiments have been run with MSPLICE1 using a distributed scheduler which implements a dynamic balance of tasks between processors. A dynamic balance here implies that each scheduling processor places a task for processing on the queue of the processor with the least number of tasks in its queue at that time. This leads to two problems. First, the number of tasks scheduled on each processor is stored within a global data structure. Distributed dynamic task allocation promotes contention for the global data structure. Second, as the number of processors increases, each scheduler must check the number of tasks scheduled on *all* the processors. Thus, as the number of processors, n, increases, the number of schedulers contending for the global data structure increases linearly with n, and each scheduler has to look through n locations in this data structure. The overall contention for the global data structure consequently grows as n^2, underlining the need for an alternate scheme for queue selection.

3.5.2 – MSPLICE1 Performance: Multiple-Queue Static Scheduling

One solution to the problem of contention during queue selection is the enforcement of static task allocation. Under a static scheduling scheme, each processor works only on those electrical nodes that are assigned to the processor at the beginning of the simulation. The speedup curves resulting from this modification are shown in Figure 3.6. It is evident that static allocation of tasks, which removes the scheduling bottleneck, resulted in significant performance improvement. For all three benchmark circuits performance is improved when using 78 processors, although there are slight degradations (less than 3%) in speedup at 32 processors relative to the multiple-queue dynamic scheduling scheme

(compare with Figure 3.5).

#Procs	Chain	Counter	Digfi
1	1.00	1.00	1.00
2	1.87	1.84	1.74
4	3.25	3.53	3.14
8	6.05	6.72	5.88
16	10.40	11.00	9.98
32	14.85	14.20	12.62
64	20.57	18.72	14.26
78	22.82	20.95	14.50

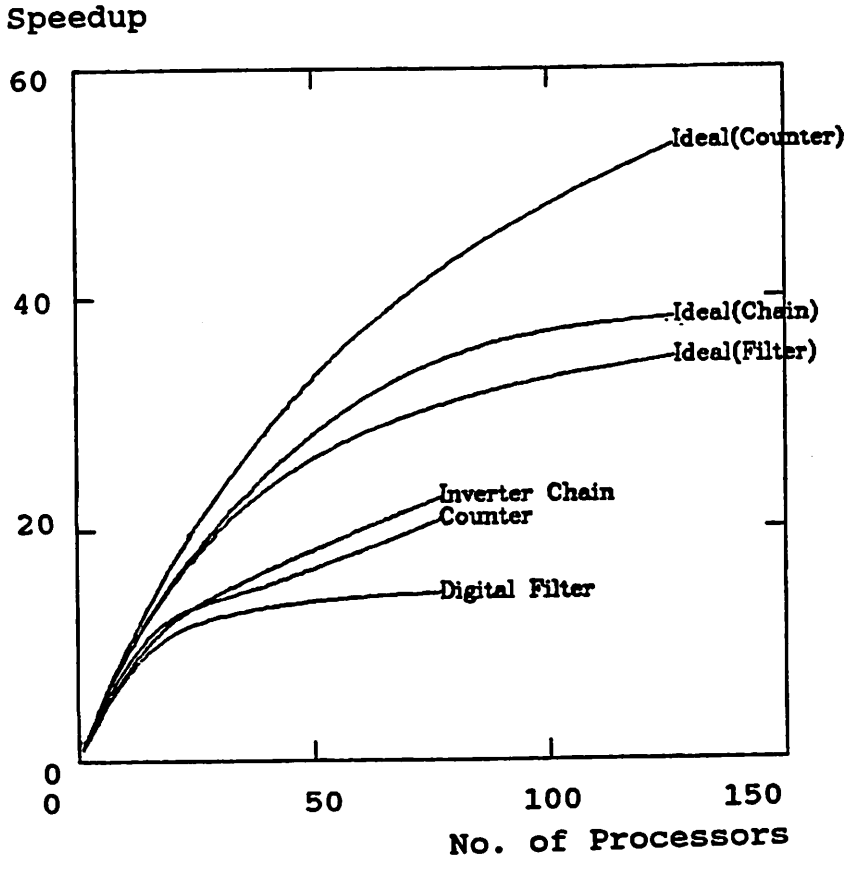

Figure 3.6. Performance curves for MSPLICE1 using static task allocation.

The results and analysis above indicate one major reason for the poor performance of MSPLICE1 relative to the ideal machine model with more than 40 processors. The ideal machine model for the scheduling process predicts better performance than occurs in practice, as it does not model the contention for the scheduling queue or the overhead of the scheduling process. It is evident that the removal of the scheduling bottleneck, by the implementation of the static task allocation scheme, causes significant improvement of performance on a large number of processors. The approach taken here, of static allocation of tasks, is not without problems. As the number of tasks and processors increases, inadequate load-balancing measures can lead to significant inefficiency. This may explain why performance did not improve for the digital-filter benchmark circuit as much as it did for the other two benchmark circuits, as shown in Figure 3.6.

3.5.3 – Globally Shared Data in MSPLICE1

Although the basic MSPLICE algorithm detailed in Section 3.2.2 uses only a single shared data value, namely *GlobalRemainingNets*, the actual implementation of MSPLICE1 requires the various processors to share other data, as listed below. Sharing of these data is insignificant when utilizing small multiprocessor systems, but becomes more noticeable as the size of the multiprocessor system grows.

Shared dynamic data, i.e., data that is changing during the course of the simulation, in MSPLICE1 includes the following:

(a) Current-time Queues on all processors

(b) Next-time Queue

(c) GlobalRemainingNets Counter

(d) Next-time Counter

(e) Node data structures

(f) Device data structures

In addition to the above, MSPLICE1 also uses the following variables which constitute shared static data:

(a) Addresses for all nodes

(b) Addresses for all devices

(c) Addresses for all models

(d) Addresses for all node fanin and fanout lists

(e) Addresses for all queues

(f) Model data structures

(g) Fanin and fanout list data structures

(h) List of active processors

While local copies are made of some of the shared static data on each processor, memory restrictions prevent the maintenance of local copies of all shared static data. In addition, due to the state of the Chrysalis programming system at the time MSPLICE1 was implemented, globally shared objects had to be explicitly mapped in to a processor's memory space before the object could be used. Although MSPLICE1 implemented a fast mapping system, the mapping had still to be performed thereby adding to the cost of accessing remote shared memory.

3.6: The MSPLICE2 Program

In order to investigate the effect of load-balancing and memory distribution, a single queue dynamic-scheduling scheme has been utilized in the new implementation of the MSPLICE algorithm, called MSPLICE2. MSPLICE2 has been implemented within the BBN Butterfly Uniform System programming environment. In Section 3.6.1, differences between the MSPLICE1 and MSPLICE2 implementations are highlighted, with emphasis on memory distribution and scheduling techniques. The performance of MSPLICE2 is described in Section 3.6.2. Profiling techniques that are now available on the BBN Butterfly are used, as described in Section 3.6.3, to study the efficiency of MSPLICE2. Timing profiles show that the IGSM is optimistic in its prediction of MSPLICE performance and that processors spend more time waiting on the queue than the IGSM predicts. In Section 3.6.4, performance of MSPLICE2 using a 64-processor BBN Butterfly with floating-point accelerators is described, showing that the relative increase in overheads results in poorer performance of the MSPLICE2 program.

3.6.1 – Differences between MSPLICE1 and MSPLICE2

Since queue selection during distributed scheduling had been identified as a signficant cause for the poor performance of MSPLICE1, the Uniform System's single-queue system appeared well-suited to MSPLICE's needs and was installed in MSPLICE2. This modification significantly reduced the amount of globally shared dynamic data as well. The increased memory available at the time of implementation of MSPLICE2 (1M byte instead of 256K bytes) also made it possible to maintain local copies of shared static data, thereby reducing contention for the interconnection network further.

In addition to the scheduling and memory distribution problems identified in the MSPLICE1 study, certain other shortcomings in MSPLICE1 were also apparent. First, the size of the benchmark circuits is limited by the size of a single processor's memory, since all simulations are run on a single processor for the purpose of comparison. Few problems can fit within the memory of a single processor and yet provide enough work for 128 processors. In another study on the Butterfly, it was determined that the cost of distributing data across the memories of 128 processors while using only one processor for computation is about 6% [37]. This is not significant when weighed against the fact that it helps prevent processor starvation. The use of the term *starvation* here refers to the state where there are not enough tasks for the available processors to work on. MSPLICE2 distributes data across the memories of all available processors, thereby allowing larger circuits to be simulated.

Another optimization used in MSPLICE2 is that of memory interleaving, which is achieved on the Butterfly by distributing global data across the memory of all the processors in the system. Memory interleaving is especially significant for frequently referenced global data, since the possibility of hot-spots due to excessive traffic at one switch port are reduced.

3.6.2 – MSPLICE2: Single-Queue Dynamic Scheduling

While the Uniform System eases distribution of shared global data and the maintenance of local copies of shared static data, the most significant modification between MSPLICE1 and MSPLICE2 is the single-queue dynamic scheduling scheme of MSPLICE2. In addition, the removal of the necessity of explicitly mapping nodes, dev-

ices, models and fanin and fanout lists into a processor's address space reduces the overhead of accessing remote data. The performance of MSPLICE2 detailed in this section is improved due to these modifications.

The performance using the MSPLICE2 implementation is shown in Figure 3.7 for the three benchmark circuits described in Section 3.4. For all three benchmark circuits, load-balancing is seen to improve performance so much so that the speedup at 32 processors here is greater than or comparable with the speedup at 78 processors using static task allocation. However, again the effects of queue contention are apparent as performance degrades or does not improve as more processors are used. It is evident from the improved performance shown in Figure 3.7 that dynamic scheduling does address the issue of load-balancing adequately, resulting in improved performance over static scheduling using only 32 processors.

#Procs	Chain	Counter	Digfi
1	1.00	1.00	1.00
2	2.00	2.00	2.00
4	4.00	4.00	4.00
8	7.87	7.97	7.96
16	13.79	15.21	14.38
32	19.55	24.16	20.41
64	21.77	30.81	25.32
101	21.85	32.22	26.63

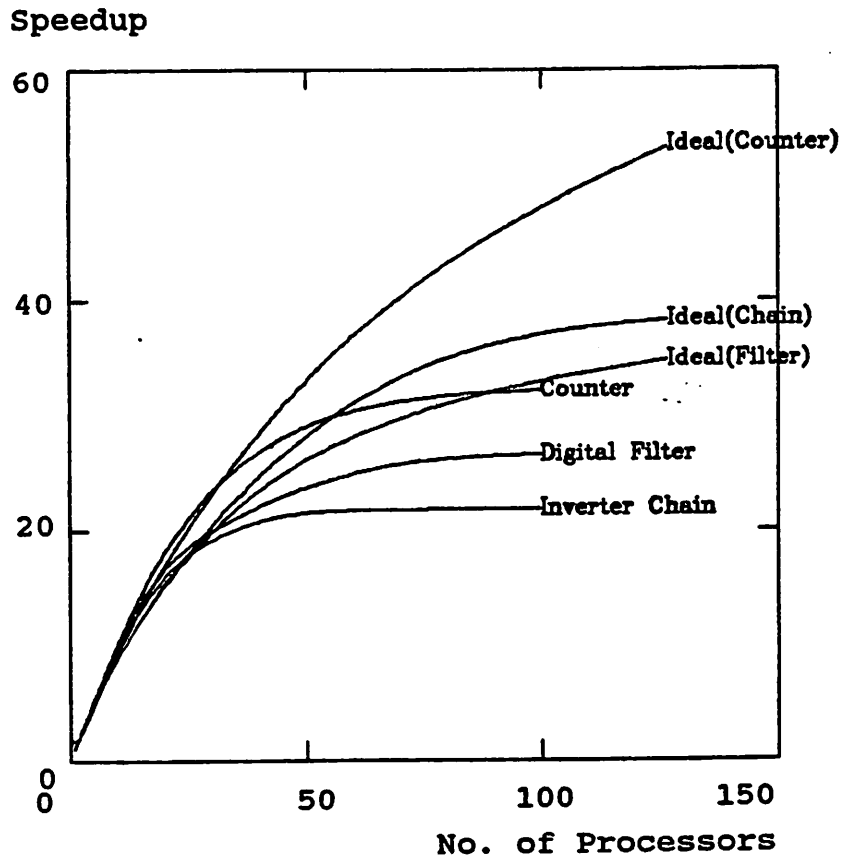

Figure 3.7. Performance curves for MSPLICE2 using single queue, dynamic task allocation.

3.6.3 – Limitations of the Ideal Gauss-Seidel Machine

It is evident from Figure 3.7 that MSPLICE2 performance, like that of MSPLICE1 using either dynamic or static scheduling, falls significantly short of the performance predicted by the IGSM for large numbers of processors. In order to determine the reason for performance using MSPLICE2 falling short of the IGSM predictions, a profiling tool developed for the BBN Butterfly [67] has been used to measure the amount of time the average processor spends waiting on the queue. Results from this profiling, shown in Table 3.3, indicate that processors actually spend *more time* waiting for work to appear on the queue than is predicted by the IGSM. This is because the IGSM, through its assumption of uniform task size, predicts that all tasks will be completed at the same time. However, in reality, tasks are of different sizes depend on the type, number and state of the devices attached to each node; as a result, processors that are assigned short tasks become idle long before those assigned long tasks and hence spend more time busy-waiting on the queue.

The IGSM model ignores the effect of the different numbers and types of devices attached to each node, which contribute to non-uniform task sizes. The profiled data in Table 3.3 does take this effect into account and indicates that the average processor spends 67% of its time waiting for work to appear on the queue, thus performing useful work only 33% of the total simulation time. This implies that with 96 processors, only 33%, i.e. 32 processors, can be fully utilized, thus providing a new ideal performance that can be expected of MSPLICE2. Similar calculations have been made for the Counter benchmark circuit for different numbers of processors, and the performance of MSPLICE2 relative to the profiled ideal (as opposed to the Gauss-Seidel Machine ideal) is presented in Table 3.4.

Subroutine	No. of calls	% of total calls
MWait	30018	67
_shift	4144	9
afdivf	2698	6
afmulf	2300	5
_offset	1389	3
afsubf	902	2
_itoe	818	2
Sundry	2534	6

Table 3.3: Subroutine-call Profile for a Typical Processor running MSPLICE2
(Simulating COUNTER benchmark using 96 processors)

No. of Processors	Utilization(%)	Eff.(IGSM,%)	Eff.(Prof.,%)
1	100	100	100
10	100	100	100
20	85	99	99
30	73	96	96
40	65	93	95
50	56	85	90
60	50	81	88
70	44	76	84
80	39	73	82
90	35	70	80
100	32	67	78

Table 3.4: MSPLICE2 Performance relative to IGSM and Profiled Ideals
(For COUNTER benchmark)

The results presented in Table 3.4 indicate that MSPLICE2 performance does approach the ideal performance achievable for the Counter benchmark circuit, when non-uniform task-evaluation times are accounted for. The fact that MSPLICE2 performance still falls short of ideal performance, with increasing degree as more processors are used, points to the effect of memory and interconnection-network contention on memory-access time, i.e., the effect of contention on task size.

3.6.4 – Performance of MSPLICE2 using Floating-Point Accelerators

In parallel processing, it is common to reduce task size in the interests of improving load-balancing, subject to the constraint that task size does not become so small that scheduling and communication overhead swamp out the improvement due to load balancing. The temptation to reduce task size in MSPLICE from a node evaluation to a model evaluation was resisted all along in anticipation of the availability of floating-point hardware, which reduces floating-point computations by an order of magnitude, thereby reducing task size considerably, since floating-point computations dominate circuit simulation.

The BBN Butterfly is now equipped with Motorola M68881 floating-point co-processors which serve to speed up the overall simulation significantly (an average factor of about 12) but simultaneously reduce task size so that scheduling and memory-access overheads become noticeable when more than 16 processors are used, as is evident from Figure 3.8. In addition to the benchmark circuits used in the MSPLICE1 and early MSPLICE2 experiments, three larger benchmark circuits have been added to this study. These include a 4-bit, 1039-node, 1692-MOSFET counter-decoder-encoder circuit

#Procs	Chain	Scdac	Counter	Digfi	ALU	Codec2
1	1.0	1.0	1.0	1.0	1.0	1.0
2	1.9	1.8	1.7	1.8	1.9	1.9
4	3.6	3.8	3.8	3.7	3.9	3.6
8	6.5	6.9	7.1	6.8	7.3	6.5
16	10.7	11.3	12.1	11.1	10.4	10.7
32	13.1	15.1	15.8	13.6	**	13.7
63	13.1	13.0	13.6	13.3	**	12.1

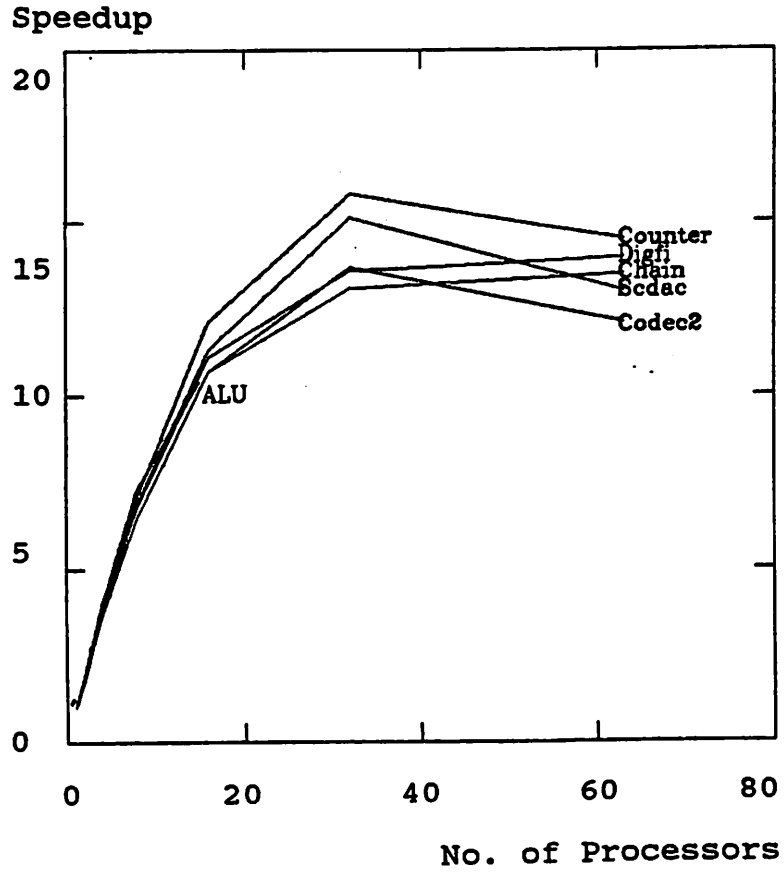

Figure 3.8: MSPLICE2 Performance using Floating-Point Accelerators
(**: Only 16 processors were available when simulating the ALU)

(Codec2), a 155-node, 416-MOSFET switched capacitor digital-to-analog converter (Scdac) and the critical path through a RISC microprocessor ALU (ALU), with 1097 nodes and 2650 MOSFET's. These benchmark circuits have been used for the analysis of MSPLICE2 only while using hardware floating-point acceleration, primarily because the availability of the acceleration hardware significantly reduced simulation runtimes for these circuits.

It is evident that once again, as with MSPLICE1 using dynamic scheduling, increasing numbers of processors (over 32) result in *degradation* of performance, i.e. simulation with 63 processors takes *longer* than with 32 processors. This indicates contention for a shared resource. Given that floating-point acceleration reduced MSPLICE2's simulation time on 1 processor from 19,010 seconds to 1269 seconds, i.e., an increase of a factor of 14.98, it is obvious that the acceleration places more strain on the interconnection network. As with the change from MSPLICE1 to MSPLICE2, this strain can be alleviated by reducing access to shared data, in this case the single queue. One solution to contention for the shared queue implemented in [61] is the use of multiple queues, such as 2 or 4 queues on a 64-processor Butterfly.

While tuning of the scheduling scheme in MSPLICE2 will continue to be required as the number and power of processors and the interconnection network increases, it is interesting to compare the absolute performance of MSPLICE2 using floating-point hardware to that of typical uniprocessors with floating-point accelerators. The performance of MSPLICE2 on a Digital MicroVAX (VAXstation II/GPX), a VAX 8800 and a 32-processor Butterfly is presented in Table 3.5 below.

It is evident from Table 3.5 that MSPLICE2 running on a single Butterfly processor with floating-point hardware is about half as fast as a Digital MicroVax workstation, while

Circuit	MicroVax	Vax8800	Butterfly	
			1-proc	32-proc
Scdac	189.2	26.02	375	24.9
Chain	510.6	68.96	1186	90.5
Counter	547.5	76.88	1269	80.3
Digfi	576.9	86.88	1341	98.3
Codec2	1238.8	194.69	2762	202.01

Table 3.5: Simulation Times (secs) for MSPLICE with Floating-Point Accelerators

MSPLICE2 running on a 32-processor Butterfly is almost as fast as a Digital VAX 8800. It is also interesting to note the effect of multiprocessor chaotic relaxation on the CHAIN and COUNTER benchmark simulations: on a single processor, the inverter chain simulation takes slightly less time than the counter simulation; however, with multiple processors, the reverse is true indicating both, the deleterious effect of ignoring Gauss-Seidel ordering, as well as the advantage of multiprocessing when there is more than a single signal path in a circuit.

3.6.5 – Parallel Processing and Iteration Counts

For SPLICE1.7 [7], which like MSPLICE is a node-based relaxation simulator, it has been noted that the effect of tight coupling (namely, floating capacitors) is to increase the number of iterations per node from 3-5 to 6-20. The recognition that tight coupling causes a node-based simulator to run slower on a uniprocessor is the reason that state-of-the-art relaxation simulators use mixed direct-relaxation algorithms to solve the circuit equa-

tions [7,9], solving tightly coupled subcircuits using direct methods. However, for a multiprocessor-based relaxation simulator, such as MSPLICE, an increased number of iterations does not necessarily translate to slower simulation, since the additional iterations can be performed on different processors. The relationship between the total number of iterations for a 10-nanosecond simulation and the number of processors performing the simulation are presented here for the benchmark circuits investigated.

It is evident from Table 3.6 that in general there is a tendency that with an increasing number of processors *less* iterations are required, i.e., the additional processors, by making up-to-date information available earlier during each timestep simulation, reduce the number of iterations necessary. It is encouraging to note that for the Scdac and Digfi benchmarks, which have the most analog nature of the benchmark circuits, the number of iterations progressively decreased, indicating that the early availability of updated information is more significant for analog than for more digital circuits. However, Scdac and Digfi are not tightly coupled circuits, hence further investigation is necessary into the performance of

#Procs	Chain	Scdac	Counter	Digfi	ALU
1	1577	14545	9876	24734	6153
2	1816	14382	9976	23651	6143
4	1848	13845	9908	23502	6127
8	1826	13506	9875	23042	5982
15	1854	12929	9758	22235	6017

Table 3.6: Total Iteration Count (Cumulative over all nodes)

MSPLICE with tightly coupled circuits.

3.6.6 – MSPLICE2 simulating tightly coupled circuits

Towards investigating the properties of MSPLICE2 while simulating tightly coupled circuits, a 395-node 723 MOSFET Sense Amplifier circuit was used. Since MSPLICE2 is a node-based simulator, each node is normally permitted a maximum of 200 iterations before MSPLICE terminates with a non-convergence error (as opposed to the 3-10 iterations that direct-method simulators permit before reducing the timestep). However, even allowing 1000 iterations per node, MSPLICE2 was unable to obtain dc convergence for the Sense Amplifier circuit, underlining the limitations of a node-based simulator using a poor initial guess. MSPLICE2 does, however, converge to a dc solution for this circuit using source-stepping techniques, i.e., by simulating certain dc sources as being switched on a few nanoseconds after the beginning of simulation.

A 5-node, 3-MOSFET boot-strapped inverter circuit, with a floating capacitor [7] was also simulated using MSPLICE2. Speedup figures and iteration counts for this simulation are shown in Table 3.7 below.

While the low speedup figures in Table 3.7 are not significant since the circuit is small and hence has a small degree of parallelism, it is significant that performance degradation can be observed with 15 processors, while for a similar-sized inverter circuit no performance degradation occurred till 64 processors were used. This indicates, as is evident from the increasing iteration count, that increasing numbers of processors degrade MSPLICE2 simulation performance. For the boot-strapped inverter circuit, it was observed that multiprocessing resulted in inappropriate ordering of node-equation solutions, thereby

#Procs	Speedup	No. of Iterations
1	1.0	982
2	1.3	1094
4	1.3	1119
8	1.3	1220
15	1.2	1361

Table 3.7: MSPLICE2 Simulation Results for Boot-strapped Inverter Circuit

increasing the number of iterations, and the boot-strapped node in the circuit often fails to converge at dc when 15 processors are used. This experiment further underlines the limitations of node-based relaxation simulators when simulating tightly coupled circuits.

3.7: CONCLUSIONS

The MSPLICE algorithm has been implemented on a Butterfly multiprocessor using 101 processors as well as on a 64-processor Butterfly with floating-point acceleration hardware. The results for a number of test circuits have been used to compare the performance of the program with the behaviour of an ideal Gauss-Seidel machine. In addition, the ability of the multiprocessor implementation to exploit the latency available in the circuit has been compared with the maximum amount of latency present, for comparable timestep control and convergence criteria.

A number of modifications have been made to the initial MSPLICE1 implementation to improve performance with a large number of processors. In particular, the distributed

multiple-queue dynamic-scheduling bottleneck has been identified and removed. The scheduling bottleneck was circumvented using static scheduling, which eliminated the contention during queue selection but does not adequately address the problem of load management. A new implementation of the MSPLICE algorithm, MSPLICE2, has been built and uses dynamic scheduling scheme with a single task queue.

The new program, MSPLICE2, uses copies of globally shared static data to reduce contention for shared memory -- this is facilitated by increased memory availability on each Butterfly processing node. MSPLICE2 uses memory interleaving to scatter dynamically varying shared data and thus avoid memory hotspots. These modifications allow the simulation of larger circuits, circumventing the starvation problem identified with earlier benchmark circuits.

The dynamically varying part of the global data structure cannot be totally eliminated, since it is necessary for communication between the cooperating processes. Given the recognition of the importance of the memory-contention problem as the number of processors increases, it is evident that effort must be directed towards minimizing both the size of the dynamically varying part of the data structure as well as the number of references to it. In MSPLICE, the *GlobalRemainingNets* counter contributes to the dynamically varying part of the global data structure.

The benchmark circuits presented here are predominantly digital, with little feedback. Traditionally, feedback paths tend to degrade performance of relaxation simulators. However, when there is a starvation problem, feedback leads to increased work. For a multiprocessor circuit simulator with a starvation problem, the increased work results in better performance relative to the uniprocessor version of the same circuit simulator. There is a

danger of viewing this *relative speedup* as an indicator of the effectiveness of the multiprocessor-based circuit simulator. Up to a point, the multiprocessor simulator may actually be effective as was seen in the case of the Scdac and Digfi benchmark circuits. However, further increases in utilization may actually be indicative more of shortcomings in the algorithm than of improved multiprocessor performance. This was evident from the boot-strapped inverter experiment.

In conclusion, it has been observed for MSPLICE that while random static scheduling works well for a few processors, large-scale parallel processing (using on the order of 100 processors) requires either more intelligent static scheduling or dynamic scheduling, the latter being the approach taken here. The single-queue dynamic scheduling is not without problems, since a single queue becomes a bottleneck as the number or power of processors increases making it necessary to use a distributed-queue system. However, for a system like the Butterfly, with a fast interconnection network, it is seen that between 16 and 32 processors can share each queue without severe contention. Finally, although load balancing varies with application and architecture, it is seen that a single circuit-node evaluation is an effective task size for the prototype simulator, MSPLICE2, on the Butterfly when using software floating point. With special-purpose hardware for accelerating floating-point operations, a single equation-solution task begins to lead to contention with about 64 processors, since the reduced task computation time is now comparable with the interprocessor communication time. This problem can be solved either by reducing the interprocessor communication time by increasing the number of queues or by increasing the task computation time by clumping a number of nodes together to form a single task. Floating-point hardware also significantly reduces simulation time, resulting in MSPLICE2 performance on a 32-processor Butterfly that approaches the performance of a VAX 8800.

CHAPTER 4

PARALLEL DIRECT-METHOD CIRCUIT SIMULATION

It is evident from the MSPLICE experiment described in Chapter 3 that circuit simulation is an application well-suited to parallel processing. However it was observed in Chapter 3 that a node-based relaxation simulator makes many iterations when simulating tightly coupled circuits; for tightly coupled circuits, there are often a small number of inter-dependent nodes that are active and it has been observed that an increasing number of processors appears to *increase* the number of iterations required to converge rather than decrease them. Hence, further research into parallelizing practical circuit simulators is necessary. Parallel subcircuit-based relaxation-direct simulators, which decompose a circuit into a number of loosely coupled subcircuits, using relaxation to solve between the subcircuits and direct methods to evaluate the tightly coupled nodes within each subcircuit, retain the decoupling intrinsic to an iterative method and would seem to be natural successors to MSPLICE. However, as experiments with PRELAX [10] and PSPLICE [43] have shown, efficient parallelization of subcircuit-based relaxation simulators is limited by load-management problems due to subcircuits with nonuniform sizes. Nonuniform subcircuit sizes result in irregular task sizes, which in turn cause load-balancing problems as some processors become idle while others continue to work on large subcircuits. Hence, parallel direct-method simulators are needed for two reasons: to improve performance of programs such as PRELAX and PSPLICE, as well as to obtain an alternative parallelization method.

The best uniprocessor algorithms for circuit simulation have been optimized for speed, so it is advisable that initial attempts at parallelizing simulation use an intuitive parallelization of the best serial algorithm for the application. If the best serial version

parallelizes well, it is unlikely that a better parallel implementation will be found. For example, an $O(nlog(n))$ algorithm for sorting will perform better even in parallel than an $O(n^2)$ algorithm, where n is the number of elements being sorted, unless the overhead (which contributes as the constant of proportionality) of the former parallel scheme greatly exceeds that of the latter. Note that if the best serial algorithm parallelizes poorly, e.g., with significant overhead per task, it *is* possible that the parallel version of a less-efficient serial algorithm will perform better.

In this chapter, an experiment in parallelizing the best-known algorithms for direct-method circuit simulation is described. In Section 4.1 the direct-method algorithms described in Chapter 2 are recapitulated, identifying the most CPU-time expensive phases of direct-method transient-analysis simulation and proposing straightforward schemes to parallelize these phases. In Section 4.2 the test-bed Sequent Balance multiprocessor is described. Parallelization of linearization and linear-equation solution, the Newton-Raphson loop phases that dominate circuit-simulation time, are described in Sections 4.3 and 4.4, respectively. In Section 4.5, the results of this experiment are summarized, indicating the potential for parallelization of different phases of the best uniprocessor direct-method algorithms.

4.1: Analysis of Direct-Method Algorithms for Parallelization

The direct-method circuit-simulation transient-analysis process, described in Chapter 2, is repeated here: Start with a set of first-order nonlinear ordinary differential equations (ODE's) modelling the circuit. Based on the smallest estimated timestep, integrate the nonlinear ODE's using a stiffly-stable integration method, at each simulation time-point.

This yields a set of nonlinear algebraic difference equations, which are solved using an iterative Newton-Raphson technique. Each iteration involves replacing all nonlinear elements by linear companion networks. Finally, solve the resulting set of linear equations using sparse-matrix LU-decomposition.

The process description above indicates the three dominant phases of direct-method simulation, all of which are candidates for parallelization. First, determination of the fastest-changing circuit variable can be performed in parallel, yielding the time-step. Next, multiple processors can concurrently linearize nonlinear devices and store the Jacobian coefficients. Finally, the set of sparse, linear equations can be solved co-operatively. A pictorial representation of this basic parallel algorithm is shown in Figure 4.1.

Although the approach to parallelization presented above produces obvious synchronization bottlenecks between the three different phases, if the phases themselves display significant parallelism, efficient load balancing during each phase should keep the loss in parallelism due to end-effects low. An *end-effect* refers to the degradation in parallelism that occurs towards the end of a computation, due to there being less tasks than processors available, and hence not enough work to keep all the processors busy. This underlines the importance of exploiting fine-grained parallelism, since small tasks would cause minimal degradation of load balancing due to end-effects. The cost for using a fine granularity is that of increased overhead for scheduling. Thus, a multiprocessor with a small scheduling overhead will handle small tasks inexpensively, thereby increasing the potential for effective load balancing.

From previous studies of direct-method circuit simulators [4, 6, 19], it is apparent that most of the transient-analysis solution time is spent in the Newton-Raphson phase of the

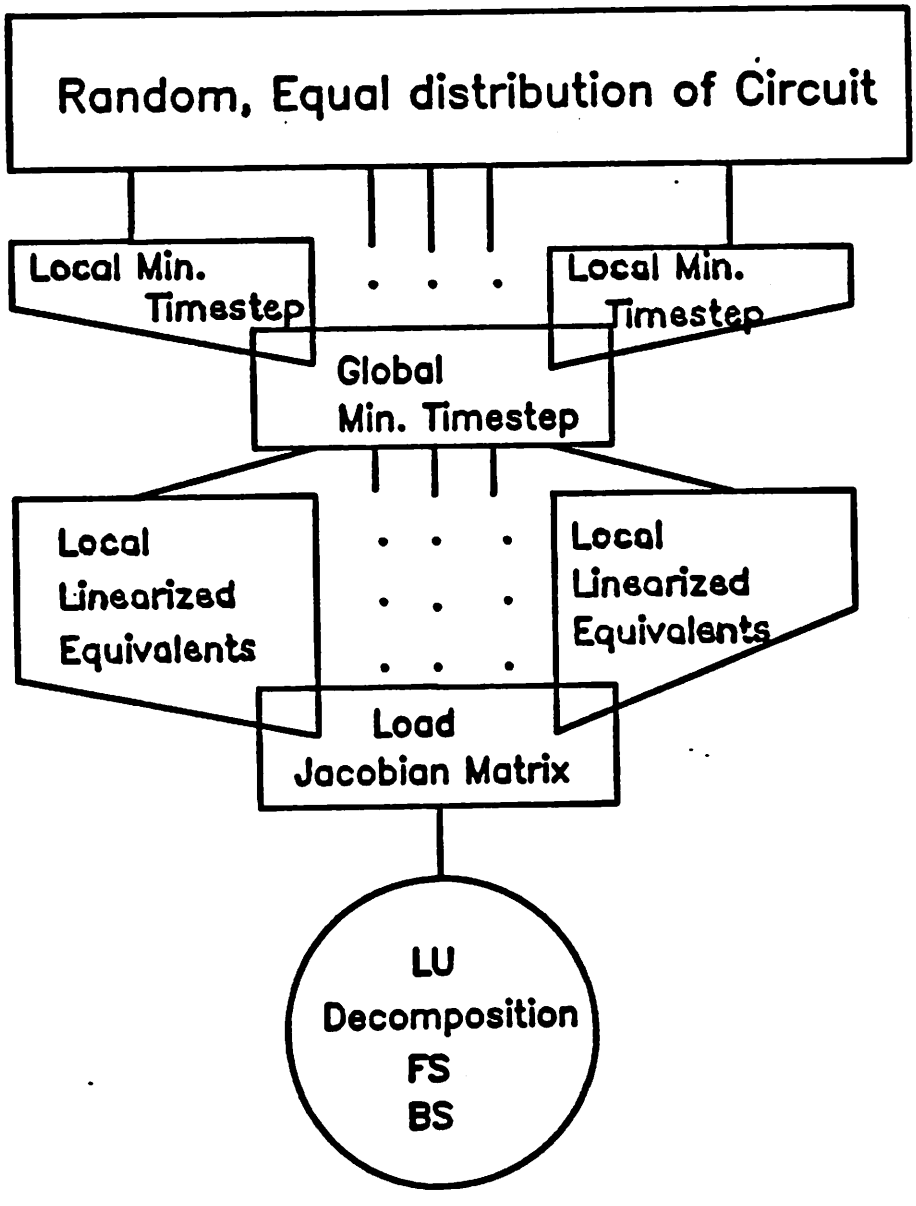

Figure 4.1. Parallel Direct-Method Circuit-Simulation Algorithm

direct-method algorithm. The two primary components of the Newton-Raphson method are the linearization and the solution of the resulting set of linear equations. Linearization involves computing linear companion models for the nonlinear devices in the circuit. The time for linearization is $O(n)$, the number of nodes in the circuit, while the time for the solution of the system of linear equations is $O(n^{1.2-1.5})$ [6,24] using optimized sparse-matrix solution techniques. Thus, as circuits with increasing numbers of nodes are simulated, linear-equation solution time begins to dominate over linearization time. The value of n for which the linear-equation solution becomes the dominant factor in determining solution time depends on the complexity of the device models, the efficiency of the sparse-matrix solution algorithms and computer memory-access and floating-point computation times.

The third step of importance in the circuit simulation is the determination of the integration timestep for the differential equations. This step involves computation of the rate of change for all the circuit variables. Once these rates have been calculated, it is necessary to determine the fastest among them. The integration timestep for the ODE's is chosen such that the expected error for the fastest-changing signal does not exceed a user-specified maximum error. Timestep selection is parallelized in two steps: circuit variables are first divided equally between processors which compute the variables' respective rates of change; simultaneously, each processor keeps track of the maximum rate of change among the variables it has solved for. As processors become free, they compare between each other to determine the globally fastest changing variable.

The parallel circuit simulator, PDSPLICE3 (listed in Appendix A), implemented on the Sequent is based on the direct-method circuit simulator used in PSPLICE3 [43]. This

has been done for two reasons: first, the parallel direct-method simulator is designed to be modular, so as to augment easily the parallelism that a parallel relaxation simulator is able to exploit by decoupling loosely coupled subcircuits; secondly, PSPLICE3 has the infrastructure to exploit the circuit property of latency, i.e., to bypass simulation of those parts of the circuit that are not active at a particular time. Since direct-method simulators can exploit circuit latency, albeit to a lesser degree than in relaxation simulators [6], it is intended that the parallel direct-method simulator make use of latency.

4.2: The Sequent Balance B8000

The Sequent Balance B8000 has been selected as a test-bed multiprocessor because it is easy to use for initial development of parallel processing algorithms due to its UNIX-like operating system, DYNIX. Indeed, both PRELAX and PSPLICE have been developed on the Balance, providing further incentive to use the Balance. Further, use of the Balance allows for easy comparison with uniprocessor direct-method circuit simulators, since a uniprocessor program compiles and runs on the Balance without modifications. However, performance of a bus-based architecture degrades as the number of processors increases, and the extensibility of this architecture for large-scale systems is still unknown.

The Balance machine available at the time of this research had eight general-purpose, 32-bit microprocessors that share a common high-bandwidth bus, as shown in Figure 4.2. At the time of this research, the largest Sequent Balance (B21000) system available has 30 processors and further extensions with more processors are planned. Processors are tightly coupled, share a single memory system, and are identical in all respects. Exclusive access to shared data structures is provided by hardware spinlocks. The programming environ-

ment is that of a proprietary operating system, known as DYNIX, which strongly resembles UNIX. The following sections provide a brief overview of the architecture and programming environment of the Sequent Balance B8000, emphasizing those features that are important when developing parallel algorithms to run on the machine.

4.2.1 – Architecture of the Sequent Balance B8000

Each processor on the Balance B8000 is a National Semiconductor Series NS32032 CPU. As the NS32032 was not designed to be part of a multiprocessor system, extra circuitry is required to support the microprocessor's interaction with its environment. The support circuitry includes a System Link and Interrupt Controller (SLIC) chip, which handles communication between the CPU and the rest of the system; 8 Kbytes of local RAM, for frequently accessed kernel code, and 8 Kbytes of cache RAM, for blocks of most

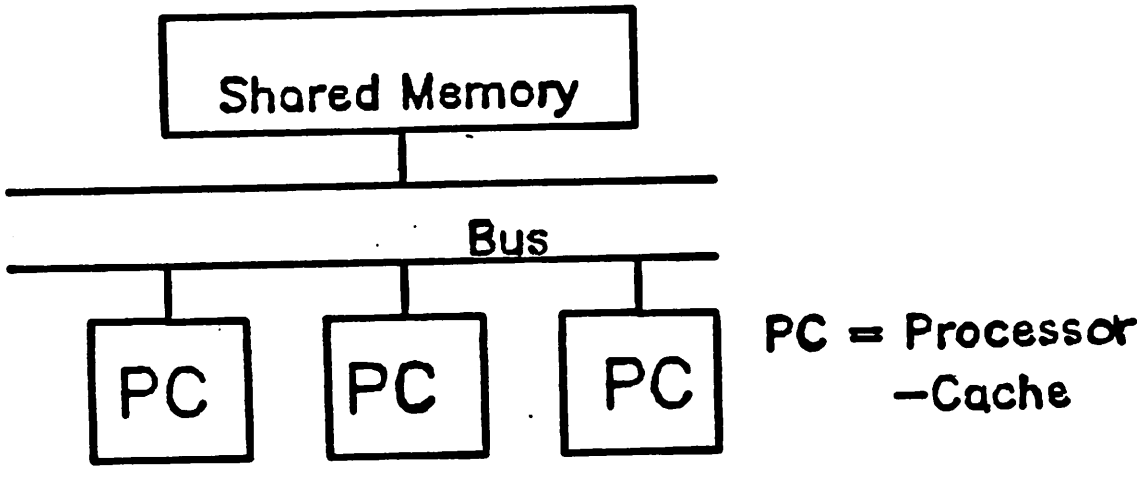

Figure 4.2. The Sequent Balance B8000

recently used system memory; an NS32082 Memory Management Unit, that controls access to memory, and supports virtual addressing; and an NS32081 Floating Point Unit for floating-point operations.

The Balance is built around the SB8000 bus, which links the 8 processors, 14 Mbytes of system memory and I/O subsystems. 32-bit data and 28-bit addresses are time-multiplexed on the bus, which has a channel bandwidth of 40 MB/s and supports data transfers at up to 26.7 MB/s. Bus operations are pipelined, so that the bus is available for transactions between the making of a request and the system memory's response to the request. The system memory can respond to a read request in 3 cycles (300 ns) and to a 4- or 8-byte write request in 2 cycles, and is capable of correcting single-bit errors and detecting double-bit errors. In addition, when two equal-sized memory modules are available, alternate 8-byte address blocks are interleaved between the two modules.

As each of the Balance processors has a cache, and all the processors may work on the same data, a *cache coherency scheme* is used to ensure that all the processors use the correct data [74]. The Balance uses a *Snoopy Cache* [75,76] consistency scheme, which operates as follows: when a processor writes into its cache, an update message is sent to the system memory as well; meanwhile, all the other processors monitor the bus, and those that recognize the updated data as residing in their own caches, update their copies of the data.

As mentioned above, the Balance contains a System Link and Interrupt Controller (SLIC) bus which is a 1-bit data path in the backplane that connects all the SLIC's in the system. The processors use the SLIC bus to ensure that only one processor can access a shared data structure, such as a lock, at a particular time. The SLIC bus provides an

efficient and easy way of ensuring mutual exclusion between the processors.

4.2.2 – Programming environment on the Sequent Balance B8000

The parallel programming environment on the Balance runs within the UNIX-like DYNIX operating system. The various operations that DYNIX provides for parallel processing are detailed in the following section. Parallel processing relies on five basic elements that single-process programs do not, namely: process creation and termination, creation of shared memory, scheduling of tasks between processes, interprocess communication, and inter-task synchronization and mutual exclusion. Operations for each of these elements is briefly described below.

To create a new (child) process, which is a duplicate of the old (parent) process, DYNIX provides a *fork()* operation. Since creation of a new process is relatively expensive (55ms on the Balance), all new processes are created at the beginning of the program and terminated, through an *exit()*, only after the entire simulation is completed. Shared memory allocation, also performed at the beginning of the program, is facilitated by the command *p_shmalloc()*, which facilitates inter-process synchronization. Among process synchronization tools, the *barrier* is particularly useful because it smooths out the wait at a synchronization point.

Debugging parallel programs is often a tedious task, since different executions of a program will probably result in different execution orders. As a result, it is difficult to duplicate the set of circumstances that exposed a particular program bug. To assist in the endeavour of identifying bugs in parallel applications, the Balance is equipped with *PDbx*, a parallel version of the Unix program *Dbx*. Prime among *PDbx*'s features is the ability to

open up windows, control and monitor progress of all the processes simultaneously, which enables a user to observe the changes in the program as and when they occur. This is a useful tool.

Performance measurement tools are essential when attempting to parallelize an application. To this end, the Balance is furnished with an impressive suite of tools, including the trace program *ptrace*, the *monitor* program and profilers *prof* and *gprof* [39,74].

4.3: Parallel Model Evaluation

Model evaluation, the linearization phase in circuit simulation, has been previously identified as a good candidate for parallel processing. This section details the parallel model-evaluation algorithm used, and the performance of the algorithm on an eight-processor Balance.

4.3.1 – Parallel Model-Evaluation Algorithm

The linearization step of the Newton-Raphson algorithm is easily and efficiently parallelizable: once the node voltages for a particular device are known, evaluation of the linear companion network for the device is performed independently of all the other devices in the circuit. The scheme used is as follows: the devices in the circuit are partitioned into approximately equal-sized groups of devices called *tasks*, which are placed on a queue at the beginning of the linearization; when free, each processor takes a task off the queue, evaluates the linearized equivalents for all the devices in the task and loads the coefficients in the Jacobian matrix. This process is continued until all the tasks have been removed from the queue.

Since the Jacobian matrix is in shared memory, loading of the matrix involves inter-process synchronization to ensure atomic additions to the matrix. The synchronization is achieved through the use of row-locks, i.e., a lock for each row of the matrix, which are locked prior to each addition and unlocked after the addition is completed. Since the time for locking, adding to a matrix location, and unlocking, is small compared to that of a typical transistor model evaluation, the synchronization overhead is not significant for transistor evaluations. Further, the circuit devices are divided equally between processors to ensure efficient load-balancing.

It is evident that in order to exploit the parallelism available during the linearization phase, the model-evaluation task time should be significantly longer than the time to load the corresponding Jacobian matrix locations. The probability of processors colliding while trying to load the shared matrix increases with the ratio of the matrix loading time to the model-evaluation time. For this reason, and to reduce scheduling overhead, it is desirable that models that can be evaluated relatively inexpensively, like nonlinear resistors and capacitors, be clumped together to form large-grained tasks. For the current implementation, devices are clumped together to form tasks with estimated computation times of 5 *milliseconds*. The estimated computation times used for devices are listed in Table 4.1 [43].

For small circuits, where model evaluation dominates total circuit-simulation time, parallel model evaluation alone speeds circuit simulation up significantly. Thus, on the Balance, a multiprocessor with a small memory on which only small circuits can be simulated, parallel model evaluation is sufficient to speed up the overall simulation time.

Device	Time (μs)
MOSFET	1890
Diode	580
Resistor	280
Capacitor	140

Table 4.1 Device Model-Evaluation Times on the Sequent Balance B8000

4.3.2 – Performance of Parallel Model Evaluation

Speedup curves for parallel model-evaluation are shown in Figure 4.3 for three benchmark circuits, displaying the near-ideal efficiency of the parallelization scheme. The benchmark circuits are a Programmable Logic Array (PLA) with 116 MOSFET's and 65 nodes, a Random-Access Memory (RAM) with 277 MOSFET's and 149 nodes, and a Digital-to-Analog Converter (DAC) with 416 MOSFET's and 155 nodes. Since all the processors work concurrently and independently on the linearization, the ideal speedup factor is nearly equal to the number of processors and is represented by the unit-slope line in Figure 4.3. The reduced parallelism due to *end-effects*, at the beginning and end of the parallel algorithm when processors outnumber tasks, is negligible since there are many more tasks than processors for most of the linearization phase (46, 138 and 109 tasks for the PLA, DAC and RAM respectively).

For small circuits, where model evaluation dominates transient analysis, parallelization of model evaluation alone makes circuit simulation noticeably faster. Table 4.2 shows the speedup in overall simulation time due to parallel model evaluation. Note that, in accordance with Amdahl's law [77], maximum speedup is limited by the unparallelized part of

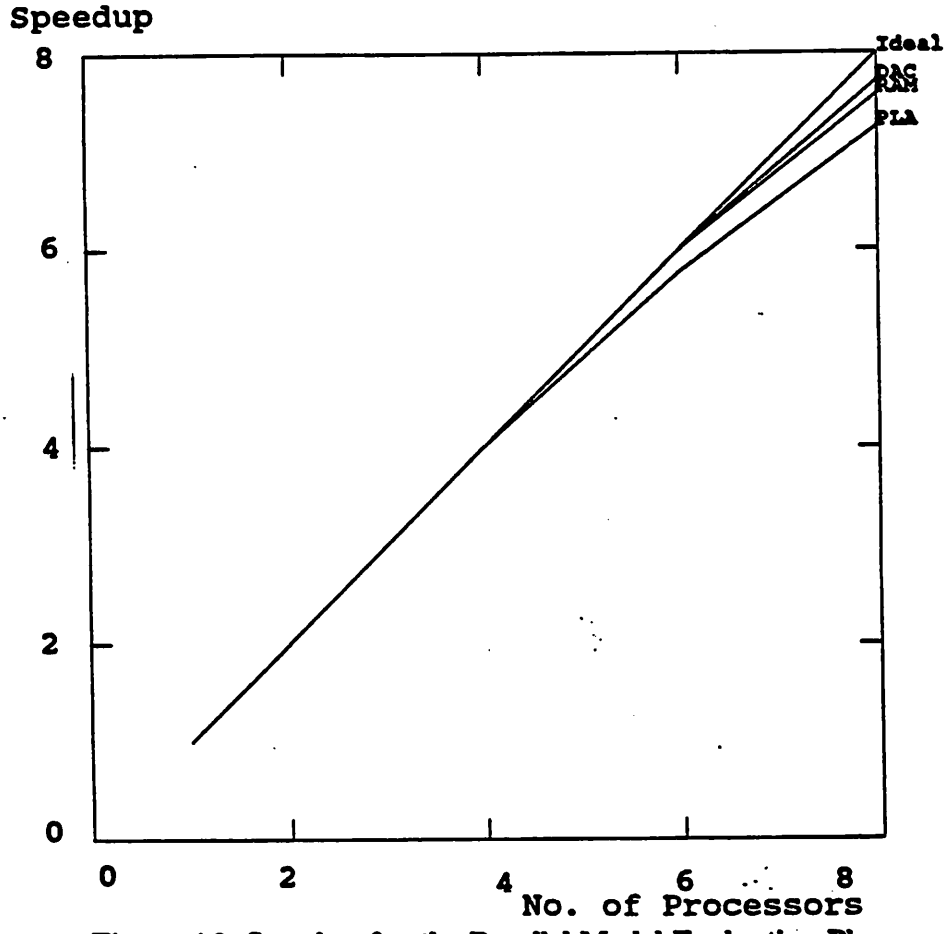

Figure 4.3. Speedup for the Parallel Model-Evaluation Phase

# of Procs	PLA		RAM		DAC	
	actual	ideal	actual	ideal	actual	ideal
1	1.00	1.00	1.00	1.00	1.00	1.00
2	1.57	1.57	1.48	1.50	1.57	1.57
4	2.18	2.18	1.91	2.01	2.17	2.17
6	2.49	2.51	2.10	2.26	2.48	2.48
8	2.60	2.72	2.17	2.41	2.65	2.68

Table 4.2: Simulation Speedup due to Parallel Linearization

the total simulation time (between 28 and 33% for the benchmarks), and hence the maximum speedup in all cases is under a factor of 3, in spite of the high efficiency of the parallel model evaluation.

As the number of processors increases to the hundreds or thousands, it is expected that algorithms that work on smaller multiprocessors will break down due to contention for the shared resources in the system, namely the bus and the memory. In addition, the shared single queue that is used in this algorithm will provide a potential bottleneck. It will then become necessary to use clusters of processors to solve smaller parts of the problem, and then solve between the clusters. The current approach prepares for this transition by arranging the problem into clusters with re-definable size. This approach is manifest in the algorithms described above, and it is expected that these algorithms will continue to be efficient as larger multiprocessing systems are utilized, albeit with multiple queues.

4.4: Parallelizing Linear-Equation Solution

It was stated in Chapter 2 that linear-equation solution time dominates overall simulation time for large circuits (over 3000 nodes). Unfortunately, parallelization of linear-equation solution time is difficult for circuit simulation, as is shown in Section 4.4.1. An efficient parallel Gaussian-elimination algorithm for full matrices is described in Section 4.4.2 and is used as a basis for the proposed parallel sparse-matrix solution algorithm. Performance of the sparse-matrix solution algorithm is detailed in Section 4.4.3.

4.4.1 – Sparsity considerations

With a full matrix, it has been shown [61] that near linear speedup is achievable for large matrices using Gaussian elimination on a Butterfly with up to 128 processors. It is observed that a speed-up factor of between 110 and 115 can be achieved for 400×400 matrices. Although back substitution has not been shown to achieve a speed-up factor greater than 80 [61], the overall linear-equation solution time, which is dominated by the triangularization phase, is accelerated by a factor greater than 105, implying a highly efficient parallelization. However, the situation is more complicated for sparse matrices.

When solving sparse matrices, in order to perform a small number of matrix operations, it is necessary to retain the matrix sparsity while picking pivots and eliminating lower triangular elements. Thus, the effectiveness of a sparse-matrix technique is measured by its ability to minimize the number of *fill-ins* into the sparse matrix, where a *fill-in* refers to the conversion of a matrix zero element that becomes nonzero during the elimination process.

As described in Chapter 2, standard direct-method circuit simulators use the Markowitz criterion [54] to pick a pivot. The Markowitz algorithm is a greedy algorithm that picks, at any stage, the pivot element that will give the least number of fill-ins if the pivot is used for elimination at that stage. The Markowitz criterion for pivot selection associates a worst-case count of expected fill-ins with each potential pivot. This count is the product of the number of nonzero elements in the pivot column and the number of nonzero elements in the pivot row, excluding the pivot element itself. The Markowitz algorithm is sequential by nature, requiring each pivot to be selected before the next one can be chosen. For most circuits, once the pivoting order has been selected at the beginning of the tran-

sient analysis, re-ordering is rarely necessary. Thus, efforts are concentrated towards parallelizing the triangularization and back substitution of a matrix with a known pivot order.

4.4.2 – The Pivot Dependency Graph (PDG) technique

A study of different pivot-selection algorithms and their respective concurrency potentials has been made, using dataflow dependence-graph analysis [78]. The pivot-dependence graph (PDG) is defined as a graph that displays pivots as nodes and dependence along the arcs and is similar to the parallel linear-equation solution graph proposed in [45]. Dependence indicates that there are certain elements in the row of the child pivot that are not known until the parent pivot has been used for elimination; hence, the child pivot cannot begin elimination until the parent element has completed its elimination. The root element in this graph, the topmost node, indicates a pivot that is not dependent on any other pivots and can be eliminated at the beginning of the linear-equation solution. An example matrix is shown in Figure 4.4(a), where alphabets denote pivot (diagonal) elements, 0's denote zeroes and X's denote off-diagonal non-zeroes. The corresponding PDG for the example matrix is shown in Figure 4.4(b).

Given a PDG, dynamic scheduling was first investigated as a means of data-driven ordering of the solution. With dynamic scheduling, a task can be performed as soon as the task's pivot becomes a root in the PDG. However, since the PDG is known at the beginning of transient analysis, dynamic scheduling introduces an unnecessary overhead in terms of locking and unlocking a globally shared queue and was discarded on that basis.

Given the unchanging nature of the PDG, static scheduling is an effective means of overcoming scheduling overhead without compromising load balancing. Indeed, in [61],

```
a x x x x o o o o o o
x b x x x o o o o o o
x x c x x o o o o o o
x x x d x o o o o o o
x x x x e o o o o o o
o o o o o f x o o o o
o o o o o x g o o o o
o o o o o o o h x o o
o o o o o o o o x i o o
o o o o o o o o o o j x
o o o o o o o o o o x k
```

Figure 4.4(a). Example Sparse Matrix

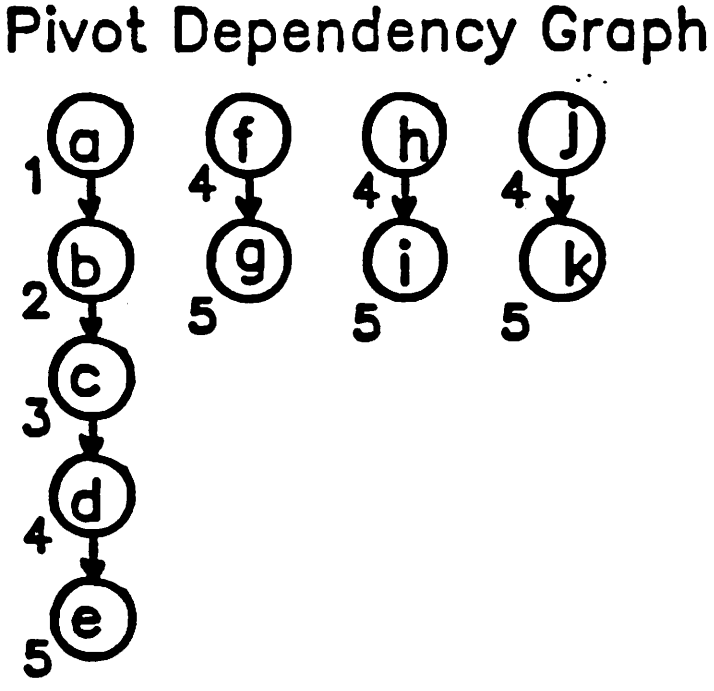

Figure 4.4(b). Pivot Dependency Graph (PDG) for Example Matrix in Fig.4.4(a)

static, sequential scheduling has been used effectively for full matrices, which have a large number of elimination tasks per pivot; hence, task-computation time greatly exceeds communication and scheduling overhead. However, for circuit matrices, with typically 3 nonzero elements per row or column, sequential scheduling of pivots does not effectively utilize a large number of processors.

For a sparse matrix, the PDG often has several root elements. A totally static way of exploiting parallelism is to schedule tasks according to their distance from the root of the PDG, thereby ensuring that all pivots at a particular level are eliminated before the next level down is started on. This works well for PDG's with multiple paths of approximately equal length. However, if one path is much longer than the rest, it is possible to have less crucial tasks, from the shorter paths, being performed by processors that would be more effectively used on tasks in the critical path. An alternative scheduling scheme is thus required.

The above shortcoming can be circumvented by assigning pivots priorities according to their distance from the *bottom* of the PDG. Now, tasks are taken off queues in order of decreasing priority. This scheme is *dynamic* and allows more critical pivots to pre-empt less critical ones for limited processing resources, but utilizes the known PDG to *statically* assign priorities. Unlike totally dynamic scheduling, there is no difficulty of having to insert tasks in the middle of a queue since each priority level is assigned a unique queue.

The example in Figure 4.5 illustrates the use of the prioritized queue system. In this example, it is assumed that 2 processors are available. Figure 4.5(a) shows the scheduling order that dynamically results when tasks are not prioritized according to their position in the PDG. It is evident that less-important tasks (h and j) are superceding the more-

important task (*b*), thereby holding up other tasks in the critical path of the computation such that the total decomposition takes 5 time units. In Figure 4.5(b), where the resulting order due to scheduling using task prioritization is shown, it can be seen that the more-important task (*b*) *pre-empts* the less-important tasks (*h* and *j*) and thereby reduces the matrix decomposition time to 4 time units.

The scheduling scheme proposed is quasi-dynamic. It exploits static scheduling by using the known PDG, yet dynamically permits high-priority tasks to pre-empt less important pivot eliminations. The scheme does not address the issue of task granularity for load balancing, i.e., subdividing a pivot task into individual row tasks, since the task size is already relatively small. As the number of fill-ins increases, the matrix for elimination becomes more dense, increasing the number of tasks per pivot elimination. The resulting increased task size would make it imperative that large tasks be further subdivided into single or clusters of row-elimination tasks for finer load balancing. This approach can be used in the event of load balancing significantly degrading performance.

The parallel PDG technique has been implemented [79, 80] within the body of the SPARSE package [81]. As part of the initial *OrderAndDecomposeMatrix()* [81], the PDG is constructed and pivots are assigned levels from the bottom of the PDG. Then, during each call to *DecomposeMatrix()* [81], the processor decomposing the matrix places all level-1, i.e., independent, tasks on the queue and waits until the other processors have completed eliminating level-1 pivots before placing level-2 pivots on the queue, and so on. Note that this implementation keeps one processor constantly busy-waiting for the others to complete pivot eliminations. A synopsis and analysis of results obtained using this implementation of the PDG algorithm are presented in the next section.

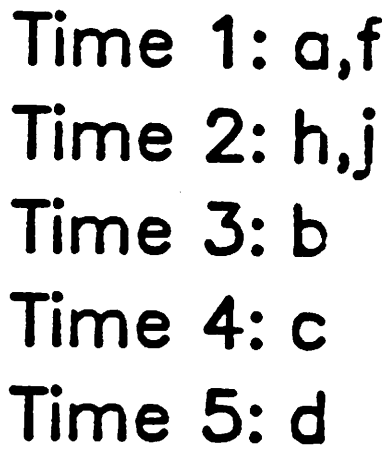

Figure 4.5(a). Scheduling without Task Prioritization

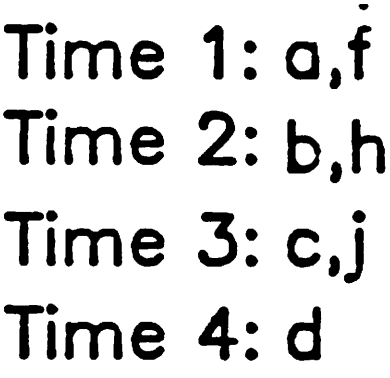

Figure 4.5(b). Scheduling with Task Prioritization

4.4.3 – Performance and Analysis

For the parallel PDG algorithm, it is possible to determine a theoretical maximum possible speed-up, $Speed_{max}$, using the following formula:

$$Speed_{max} = \frac{N_{piv}}{CP_{length}} \tag{4.1}$$

where N_{piv} is the number of pivots in the matrix, and CP_{length} is the length of the PDG critical path. From Eqn.4.1, the maximum possible speed-up for the three benchmark circuits is shown in Table 4.3 and is achievable *iff*

$$N_{proc} \geq N_{wide} + 1 \tag{4.2}$$

where N_{proc} is the number of processors and N_{wide} is the number of pivots at the widest level of the PDG.

In spite of the already small $Speed_{max}$, for all three benchmark circuits, it is observed that the parallel PDG algorithm results in significantly longer solution times for multiple processors than for a single processor. This phenomenon is due to the implementation of the algorithm as explained below.

It is observed that the PDG for all three of the benchmark circuits has a wide top, i.e., a lot of independent (level-1) pivots, but narrows as the level within the graph

Circuit	N_{piv}	CP_{length}	$Speed_{max}$
Decpla	56	17	3.3
Scdac	150	23	6.5
Cramb	139	26	5.3

Table 4.3: Maximum Possible Speedup using Parallel PDG Algorithm

increases. As a result, there are a number of tasks that can be performed in parallel at the beginning of the solution, but the triangularization process becomes essentially sequential after a stage. Once the solution reaches the point in the PDG after which there is only one pivot per level, the current algorithm is extremely wasteful since it involves one processor scheduling tasks for another processor to perform, explaining the slowing down of the solution process.

4.5: Conclusions – Performance and Extensibility of Parallel Direct-Method Solver

Although the results in Section 4.3.2 indicate that efficient parallel model-evaluation can be achieved, it is not clear that these results are extensible on a Balance-like architecture with increasing numbers of processors. In particular, it is expected that as the number of processors increases, contention for the shared resources, i.e., the bus and shared memory, will increase until communication and scheduling overhead swamp out the useful computation. Initially this effect can be countered by increasing task size, but it is evident that increased task size ultimately leads to degradation in load balancing. Hence, an architecture that distributes memory and interconnection network is required for the continued efficacy of the parallel model-evaluation algorithm used here.

The matrix-loading algorithm used in the parallel direct-method solver is random by nature, i.e., nothing is done to ensure that different processors aren't working on the same row of the Jacobian matrix (and hence competing for the same lock) or to stagger task sizes such that all processors aren't attempting to update the matrix or take tasks off the queue simultaneously. Although more sophisticated algorithms may become necessary as more processors are used, for the purposes of this study it has been concluded that the

current scheme is adequate.

In summary, a direct-method circuit simulator, implemented on the Sequent Balance B8000, serves to provide a measure of the concurrency potentials of the best uniprocessor algorithms and a metric for the efficiency of other parallel-processor circuit simulators. An intuitive parallelization scheme is presented, and it is seen that this scheme results in efficient parallelization of the linearization phase of circuit simulation, i.e., model evaluation. However, the proposed pivot-dependency graph technique does not yield a high degree of exploitable parallelism for linear-equation solution, as shown by the results in Section 4.4.2. Improved implementations of the PDG scheme are presented in Chapter 5, along with an analysis of the $Speed_{max}$ figures for larger circuits.

CHAPTER 5

PARALLEL SPARSE LINEAR-EQUATION SOLUTION

The potential for parallelization of a direct-method circuit simulator is described in Chapter 4 and reveals that parallel processors *can* efficiently exploit the parallelism in circuit-matrix linearization. To parallelize linear-equation solution, a pivot-dependency graph (PDG) scheme is proposed in Chapter 4, but results from the initial implementation indicated poor parallel performance. Results from an efficient implementation of the PDG scheme are presented in Section 5.1. It is concluded that the PDG scheme is not viable for the exploitation of large-scale parallelism in linear-equation solution, predominantly due to the limitation of its use of pivot-sized tasks. As a result, in Section 5.2, parallelism is investigated on the basis of *elemental* operations, using a finer granularity than pivots to exploit a higher degree of parallelism. Parallelization techniques applied to back substitution are described in Section 5.3, revealing that elemental techniques again have higher parallelization potential than pivot-based approaches. Results from an implementation to remove the synchronization bottleneck between the triangulation and back-substitution phases are listed in Section 5.4. Conclusions are presented in Section 5.5.

5.1: Pivot-based Parallel Linear-Equation Solution

The Pivot Dependency Graph (PDG) scheme, a technique designed to exploit as much parallelism as is available during the solution of a sparse system of linear equations ordered using the Markowitz criterion, is described in Chapter 4 (see also [79,80]). A PDG is a graph that determines the order in which pivots may be eliminated during LU-

decomposition, by establishing which pivots affect the final values of other pivots. For circuit simulation, the PDG is determined at the beginning of transient analysis and rarely needs to be changed during a simulation since the matrix order itself usually remains unchanged. Independent pivots are eliminated at the beginning of the linear-equation solution, while dependent pivots are processed as they become ready to be used. The PDG scheme yields a low degree of parallelism, merely because it deals with a sparse system of equations with typically two or three non-zero off-diagonal elements in each row or column. Indeed, for the PLA, DAC and RAM benchmark circuits described in Section 4.3, the ratios of the total number of pivots to the maximum depth of the PDG – which may be used as a measure of the parallel potential of the algorithm – are 3.29, 6.52 and 5.34, respectively.

Apart from direct inter-pivot relationships, however, it is also necessary that successor elements (below and to the right of pivots) are eliminated in the same order as they would have been eliminated by a single processor. In the initial implementation, described in 4.4, this secondary dependency was handled by busy-waiting for an element to become the leftmost uneliminated element in its row. This scheme reduced the already-small PDG parallelism limits for the PLA, DAC and RAM to 2.00, 3.57 and 3.31, respectively. Further, pivots were scheduled through the global queue, and it is observed that the overhead of queue handling, including locking and unlocking, far outweighs the useful computation of an elimination task, even for a single Balance processor. Also, synchronization between the LU-decomposition, forward and back substitution phases leads to end-effects that further degrades parallel performance. As a result of the numerous inefficiencies in the implementation of the PDG algorithm, even the small parallelism available could not be exploited. In fact, multiple processors took *longer* than a single processor for the sparse

linear-equation solution.

A new implementation counters the above problems as follows: At the beginning of an equation-solution, each processor, P_i, is assigned a unique number, Q_i (through the queue), giving the processor responsibility for a set of pivots, *PivotSet*, during that solution. The set of pivots that a processor is responsible for are determined using the formula below:

$$PivotSet = Q_i, Q_i + N_{procs}, Q_i + 2 * N_{procs}, \cdots$$

$$while\ Pivot_{number} \leq Matrix\ dimension,$$

where N_{procs} = the number of processors working on the parallel linear-equation solution. The queue is thus used only once at the beginning of each solution, greatly reducing queue-handling overhead. The busy-waiting problem is overcome by having processors set a flag when an element *can* be eliminated, but isn't because there are elements to its left that have not yet been eliminated. The processor that eliminates the previous element in the same row will notice the flag and perform the elimination. This ensures that all elements are eliminated, while relieving processors from having to wait when useful work can be done elsewhere. Finally, Gaussian elimination is used instead of LU-decomposition to remove the synchronization bottleneck between decomposition and forward solution. Results from this experiment, for the three benchmark circuits described in Chapter 4, are shown in Figure 5.1 and Table 5.1.

Although the actual speedup is still lower than the ideal speedup, performance is significantly better than with the previous implementation. Note that the ideal speedup in all cases is less than the maximum possible speedup figures (3.29, 5.34 and 6.52 for the PLA, RAM and DAC, respectively). This is because of the structure of the PDG, which is wide at the top and narrow at the bottom. As a result, a number of processors can work

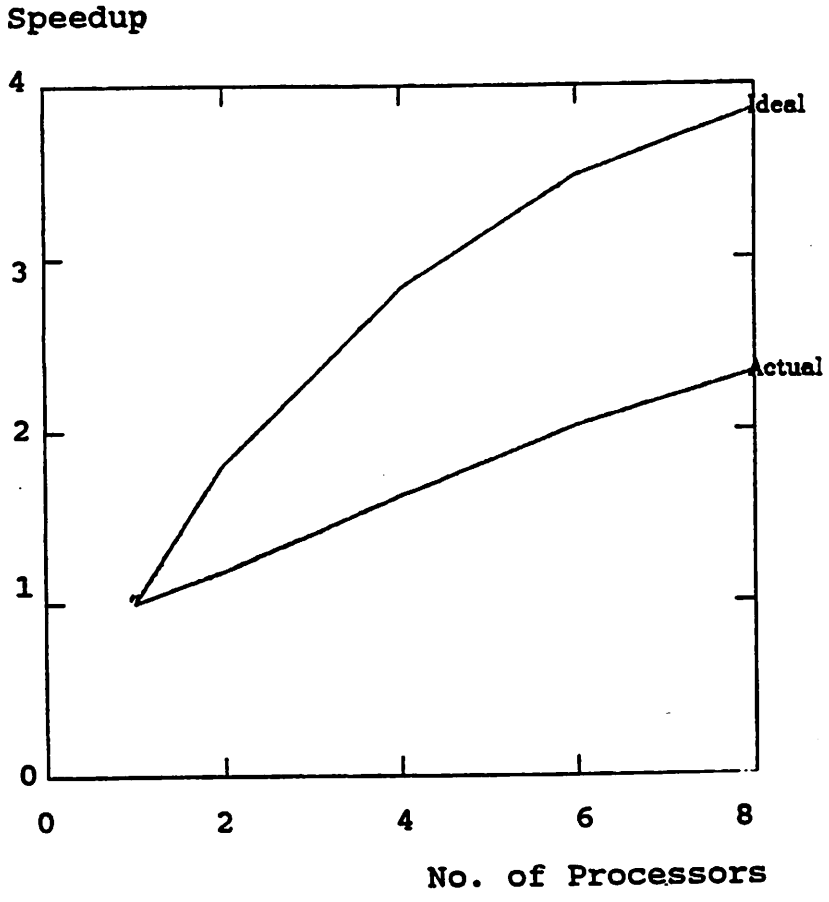

Figure 5.1: Speedup using PDG-based Parallel Linear-Equation Solution (RAM Benchmark Circuit)

# of Procs	PLA		RAM		DAC	
	actual	ideal	actual	ideal	actual	ideal
1	1.00	1.00	1.00	1.00	1.00	1.00
2	1.28	1.60	1.19	1.81	1.32	1.76
4	1.74	2.24	1.62	2.83	1.73	2.83
6	2.06	2.67	2.03	3.48	2.08	3.66
8	2.16	2.80	2.34	3.86	2.25	4.17

Table 5.1: Speedup with PDG Linear-Equation Solution

concurrently at the beginning of the linear-equation solution as shown in Table 4.3, while all three benchmarks require only one processor at the end of the solution. However, as long as the number of actual processors is less than the greatest width of the PDG, the ideal speedup is less than the maximum speedup possible. This highlights another limitation of the PDG algorithm: in order to attain the maximum parallel speedup, it is necessary to have a large number of processors available to perform triangulation at the beginning of the linear-equation solution, due to the unbalanced nature of the PDG.

5.2: Fine-grained Parallel Triangulation

Studies such as [34] indicate that there is a substantial degree of parallelism available in sparse linear-equation solution for large systems when a fine-grained parallel technique is used. In this section, the parallel potential of element-based, rather than pivot-based, operations is investigated. Although element-based parallelism is harder to exploit on a medium-granularity parallel processor such as the Balance, the investigation provides a more accurate estimate of the limit to parallel linear-equation solution. For purposes of implementation, it is possible to clump together elemental tasks, as was in fact done in [34], to overcome system scheduling and synchronization overheads.

In order to estimate element-based parallelism, a concept similar to PDGs, Row-Dependency Graphs (RDGs) are used. The RDG technique provides relationships between row operations for a matrix. For an element-based analysis, each of the pivot tasks defined by the PDG method is further subdivided into basic row operations, i.e., division for normalization and multiplication-subtraction for elimination. For Gaussian elimination, the atomic operation involves operation on all the elements in a row simultaneously, so the ele-

mental operations are merely row normalization and elimination tasks. In Table 5.2, parallelism limits using row-based and element-based computations are presented. The depth of the RDG, RDG_{depth} remains the same for both cases, and the parallelism limits calculated for row-based and element-based cases, $Speedup_{row}$ and $Speedup_{element}$, respectively, are calculated as follows:

$$Speedup_{row} = \frac{Operations_{row}}{RDG_{depth}}$$

where $Operations_{row}$ is the number of row normalizations and eliminations , and

$$Speedup_{element} = \frac{Operations_{element}}{RDG_{depth}}$$

where $Operations_{element}$ is the number of divisions and multiplication-subtractions.

The results in Table 5.2 indicate that there *is* a significant amount of parallelism that can be exploited during sparse linear-equation solution, but this parallelism is only available at a very fine granularity. While the parallelism available using row-based tasks is approximately twice that available using the PDG technique (compare Table 5.2 with Table

Circuit	RDG Depth	*Max.* $Speedup_{row}$	*Max.* $Speedup_{element}$
PLA	39	4.56	6.26
RAM	41	12.66	18.53
DAC	49	11.06	16.0
DIGFI	48	27.69	39.63
EPROM	93	32.47	51.40

Table 5.2: RDG-based Parallelism Limits
(Using row- and element-sized tasks)

5.1), the maximum speedup using element-based calculations is about three times as much. This corroborates the results published in [34,56], indicating that the parallel processor used to exploit this parallelism must be constructed such that scheduling and memory access overheads are negligible when compared to a single division or multiplication-subtraction operation.

Results from implementations of the row-based RDG algorithm are presented in Figure 5.2 and Table 5.3.

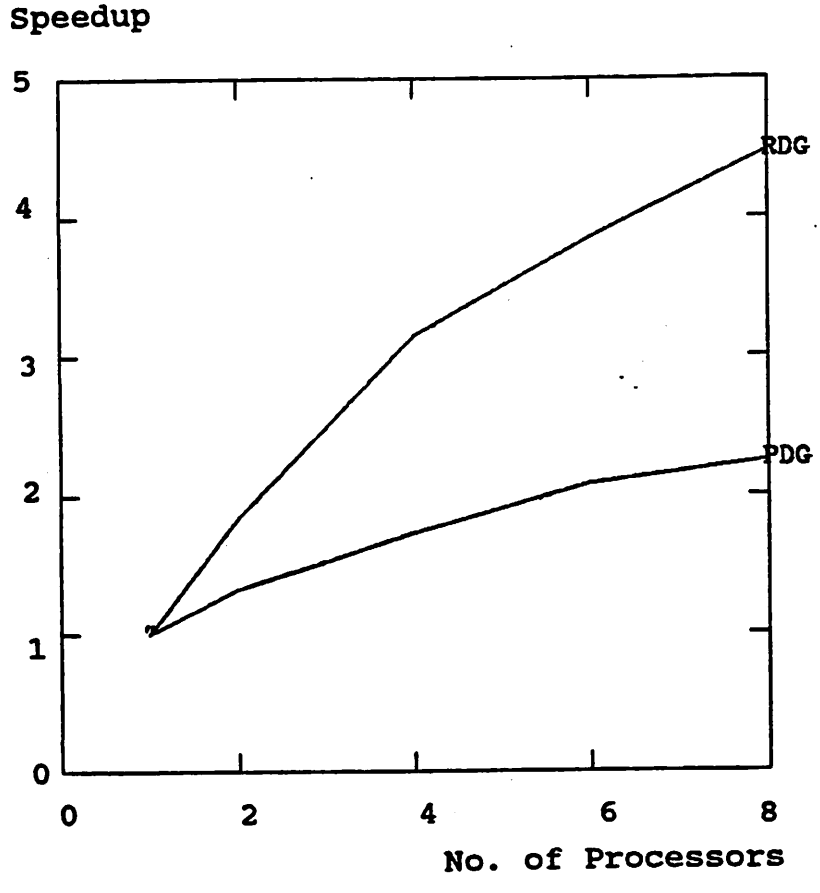

Figure 5.2: Speedup of Triangulation using Parallel RDG Methods
(DAC Benchmark Circuit)

Circuit	RDG Depth	# of Procs	$Speedup_{row}$
PLA	39	1	1.00
		2	2.61
		4	2.41
		6	2.83
		8	2.51
RAM	41	1	1.00
		2	1.73
		4	2.73
		6	3.44
		8	3.93
DAC	49	1	1.00
		2	1.84
		4	3.15
		6	3.85
		8	4.47

Table 5.3: Row-based RDG Parallel Triangulation Speedup

5.3: Parallel Back-substitution

For dense matrices, LU-decomposition is $O(n^3)$, while back-substitution is only $O(n^2)$; hence parallel performance of triangulation far out-weighs the effect of parallel back-substitution [61], especially when the matrices are large. However, for a sparse matrix, triangulation and back-substitution are both of $O(n^{1.2-1.5})$, since tracing down the sparse-matrix pointer structure and the increasing (relative) density of the sparse matrix equally affects both phases of the linear-equation solution. As a result, although back-substitution typically takes about half as long as triangulation (as is shown for the examples that follow) since half as many elements are being utilized, speedup through parallel processing depends significantly on efficient parallel back-substitution as well.

In this section, the parallel potential of back-substitution is analysed using the PDG and RDG techniques described earlier. The matrix order is pre-determined on the basis of sparsity and parallelism for triangulation. Hence, back-substitution uses the same order, and the PDG constructed merely starts from the bottom-right of the matrix and moves upwards, i.e., the opposite direction to that used in the PDG technique for triangulation. Parallelism limits using the PDG technique are shown in Table 5.4. These results are identical to those in Table 5.1, because the circuit matrix is symmetric after fill-ins have been inserted; therefore, the back-substitution PDG is merely the inverted triangulation matrix.

For back-substitution, a single row operation is actually a column operation, but is referred to as a row operation for convenience. Since each pivot-operation involves a single vector multiplication-subtraction operation for one column, the RDG is identical to the PDG. As a result, as shown in Table 5.5, the critical path through the RDG has the same length as the corresponding path through the PDG, i.e., maximum pivot-based speedup

Circuit	Equations	PDG Depth	*Max.Speedup*
PLA	56	20	2.8
RAM	139	21	6.62
DAC	150	23	6.52
DIGFI	378	33	11.45
EPROM	630	37	17.03

Table 5.4: PDG-based Parallelism Limits for Back-substitution

equals maximum row-based speedup. In addition, due to the symmetric nature of the circuit matrices, the number of elemental operations are observed to be half the number of elemental operations in Table 5.2. Also evident from Table 5.2. is the fact that, except for the largest example, EPROM, the maximum parallel speedup is slightly lower for back-substitution than it is for triangulation.

Results from implementations of the row-based RDG back-substitution algorithm are presented in Figure 5.3 and Table 5.6.

5.4: Synchronizing Triangulation and Back-substitution

Although element-based parallel-limit calculations, presented in Sections 5.3 and 5.4, indicate a reasonably high speedup potential for the two phases of linear-equation solution, triangulation and back-substitution, this potential is available only at a very fine task granularity. As a result, an efficient parallel linear-equation solver must exploit as much as pos-

Circuit	Equations	RDG Depth	Element Ops.	*Max.Speedup*
PLA	56	20	122	6.1
RAM	139	21	380	18.1
DAC	150	23	392	17.04
DIGFI	378	33	951	28.82
EPROM	630	37	2390	64.59

Table 5.5: RDG-based Parallelism Limits for Back-substitution

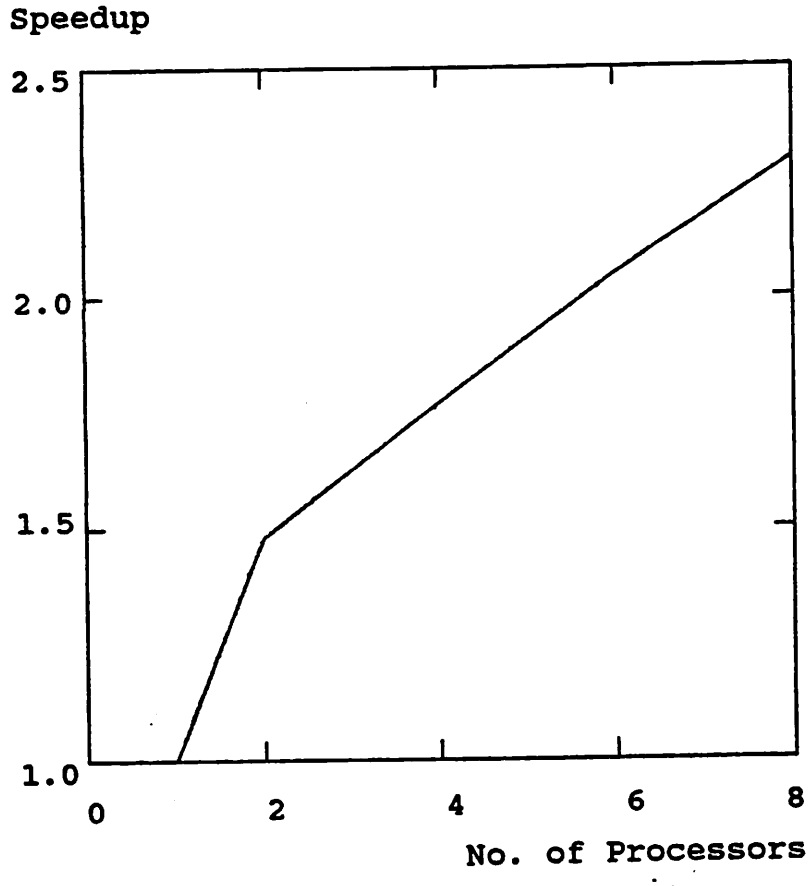

Figure 5.3: Speedup for Back-substitution using Parallel RDG Algorithm (RAM Benchmark Circuit)

Circuit	RDG Depth	# of Procs	$Speedup_{row}$
PLA	39	1	1.00
		2	2.00
		4	1.52
		6	1.99
		8	1.63
RAM	41	1	1.00
		2	1.48
		4	1.77
		6	2.05
		8	2.30
DAC	49	1	1.00
		2	1.16
		4	1.72
		6	1.99
		8	1.66

Table 5.6: Row-based RDG Parallel Back-substitution Speedup

sible of the available parallelism. This involves reducing the serial bottleneck between the triangulation and back-substitution phases, as is described in the paragraph below. In this section, the synchronization point, or serialization bottleneck, between matrix triangulation and back-substitution is investigated, and methods of reducing its length are proposed and studied.

In the implementation of the parallel linear-equation solver described so far, linear-equation solution proceeds as follows: at the beginning of the solution, each processor is assigned, through the centralized queue, a unique number, which it utilizes through the triangulation to determine which pivots, rows or elements it is responsible for normalizing or eliminating. The last processor to complete triangulation places a new set of numbers on the queue, which the processors take off the queue to determine which pivots, rows or

elements they must work on for back-substitution. Towards the end of the triangulation, some processors are idle because they have no work to do. All but one of the processors continue to be idle while new tasks are being placed on the queue. Finally, once tasks are placed on the queue, processors compete for the queue lock in order to obtain their assigned tasks. Although the costs of such a synchronization bottleneck and the associated end-effects are marginal on a system with a few processors, such as the Sequent Balance, these costs rise significantly as the number of processors which are idling or competing for the shared lock increases.

For full matrices, where all the equations depend on each other, back-substitution cannot be started until the end of triangulation, because a full matrix has a PDG that is a single chain, where pivots affect each other sequentially. Hence, if the PDG for triangulation are to be concatenated with that for back-substitution, the resulting PDG is a single chain as well, indicating that the synchronization bottleneck between triangulation and back-substitution cannot be removed by pipelining the two phases. For sparse matrices, however, the PDG has a number of parallel chains and the concatenation of PDGs for triangulation and back-substitution can provide a number of parallel paths, thereby reducing the impact of the synchronization point.

Analysis of the structure of the PDGs for the five benchmark circuits indicates that the PDG reduces to a single chain towards the end of triangulation and is therefore a single chain at the beginning of back-substitution. As a result, the concatenation of the two PDGs results in a single chain and does not remove the serial bottleneck between the two phases. However, the logical concatenation of the two PDG's provides a means of eliminating the necessity of placing flags on the queue to demarcate the beginning of back-

substitution. Instead, processors work on the same set of pivots for back-substitution as they did for triangulation. Thus, while pipelining of pivot tasks does not curtail end-effects, it does serve to reduce inter-processor contention.

While the PDG and RDG structures are essentially sequential towards the end of triangulation, there are a number of elemental operations that can be performed simultaneously. Thus, due to finer granularity, elemental parallel linear-equation solution provides an efficient way of reducing not only end-effects, but also the synchronization bottleneck. Results using elemental operations as tasks are shown in Table 5.7.

5.5: Conclusions

The analysis and results from the PDG experiment for linear-equation solution are encouraging, although the effectiveness is less than that from parallel linearization. Linear-equation solution, which dominates simulation time for very large circuits, does not display as high a degree of parallelism as does circuit linearization using the PDG scheme, but the parallelism *does* appear to increase with circuit size. The experiments conducted indicate that the organizational overhead for the PDG scheme is significant when compared with the useful work done, but part of this is due to issues of implementation of the algorithm on the Balance architecture, as is evidenced by the significant reduction in linear-equation solution time when queue usage is dispensed with.

A study of fine-grained parallelism, based on row and elemental operation and described in Section 5.2, shows that maximum parallel potential does increase significantly as the task size increases. In order to exploit this higher degree of parallelism, inter-process synchronization and task-handling overheads must be reduced significantly.

Circuit	RDG Depth	# of Procs	$Speedup_{row}$
PLA	39	1	1.00
		2	1.67
		4	2.28
		6	2.78
		8	2.39
RAM	41	1	1.00
		2	1.71
		4	2.60
		6	3.11
		8	3.43
DAC	49	1	1.00
		2	1.88
		4	3.02
		6	3.52
		8	3.57

Table 5.7: Pipelined Row-based Linear-Equation Solution
(Concatenated PDG's for Triangulation and Back-substitution)

Back-substitution is shown to have as high a degree of parallelism as triangulation using PDGs or row-based RDGs, and slightly less potential for speedup for element-based tasks.

While pipelining is an attractive way of reducing the effect of the serial bottleneck between triangulation and back-substitution, the structure of the PDGs and RDGs does not provide additional parallelism around the synchronization point, due to the serial nature of these graphs towards the end of the triangulation phase and at the beginning of the back-substitution phase. However, it is seen that element-based tasks mitigate the effect of this serialization since more tasks are available for processing immediately before and after the

synchronization point. As a result, end-effects due to improper load balancing are reduced and an effective parallelization of the otherwise-serial synchronization bottleneck is achieved.

CHAPTER 6

INTER-PHASE SYNCHRONIZATION BOTTLENECKS

Bottlenecks due to synchronization constraints between the main phases of direct-method simulation are studied in this chapter. The direct-method simulation process has been described in detail in Chapter 2, with emphasis on the main phases of the simulation. In Chapter 4, it has been demonstrated that model evaluation is efficiently parallelizable. However, as is described in Chapter 5, parallelizing the sparse linear-equation solution is not as straightforward. One possible solution to this problem is the pipelining of the two phases of the Newton-Raphson process such linear-equation solution begins before the end of model evaluation, thereby reducing idle time at the beginning of linear-equation solution. In addition, the synchronization bottleneck between linear-equation solution and convergence checking also produces a bottleneck to efficient parallel simulation.

In Section 6.1, the mismatch between the parallel potentials of the two phases of the Newton-Raphson method is examined, and an algorithm is described that pipelines these two phases to overcome the limitations of this mismatch. Pipelining linear-equation solution with convergence checking is not as simple as combining the two phases of the Newton-Raphson iteration, but certain speedups *are* possible due to pipelining and are described in Section 6.2. Further, parallel model evaluation serves to reduce convergence checking time substantially, which is also shown in Section 6.2. The combined effects of parallel model evaluation, sparse linear-equation solution and convergence checking, as well as the effects of pipelining the different phases, results in an efficient parallel direct-method solver, as is shown in Section 6.3. Also in Section 6.3 are speedup figures for PSPLICE3, with and without the parallel direct-method solver, displaying the utility of

parallel direct-method simulation within a parallel relaxation simulator.

6.1: Pipelining Model Evaluation and Linear-Equation Solution

This section deals with combining the high parallel potential and implementation efficiency of model evaluation with sparse linear-equation solution through the use of pipelining. The intent of this combination is to swamp the low parallelism in linear-equation solution with the high parallelism in model evaluation. The increase in task size, due to pipelining, and the reduced pointer-manipulation overhead, due to localization of references, further contribute to increased parallel efficiency.

In Section 6.1.1, the mismatch between the parallel potentials of the two phases of the Newton-Raphson method are presented and analysed, and a pipelining algorithm to overcome this mismatch is presented. Results from an implementation of this algorithm on the 8-processor Sequent Balance are presented in Section 6.1.2.

6.1.1 – Pipelined Algorithm: The problem and its solution

The prime reason for the poor parallel performance of sparse linear-equation solution is the low degree of parallelism available. As shown in Chapter 5, even using element-based tasks the maximum parallel speedup for a 150-equation matrix (DAC) is a factor of only about 17. By contrast, as is evident in Chapter 4, model evaluation has a parallelism potential equal to the number of devices in the circuit, if single-device tasks are used. Thus, for the same DAC benchmark circuit, the maximum parallel speedup for model evaluation is a factor of 416. This underlines the essentially sequential nature of sparse linear-equation solution and the highly parallel nature of model evaluation. As a result, the parallel algorithms described in Chapters 4 and 5 result in sufficient work to keep all the

processors busy during the first phase of the Newton-Raphson algorithm and then only enough work to keep a few processors busy during the second phase of the algorithm. This problem is especially important because linear-equation solution dominates simulation time for large circuits.

In addition to the mismatch in parallel potential between the two phases of the Newton-Raphson process described above, there is also a difference in the size of the elemental tasks used for each of the phases which affects implementation and hence performance of the parallel algorithms described in Chapters 4 and 5. In particular, the average model-evaluation task is far longer than the average decomposition task. Thus, while implementation of an algorithm which uses single-device tasks works well for parallel model evaluation, a single-pivot elimination is observed to be a task-size too small for the overhead associated with task handling and scheduling on the Sequent Balance.

The combination of the two factors described above contribute to largely differing speedups during the two phases of direct-method circuit simulation. However, the second, more sequential, phase follows the highly parallel first phase, thereby providing an effective manner to reduce the time spent during the sequential sparse linear-equation phase: *pipelining*.

The scheme proposed is as follows: once a row of the Jacobian coefficient matrix has been completely loaded during model evaluation, it can be normalized and its successor rows can be scheduled for elimination. If the row requires elimination before it can be normalized, then the processor that completes the loading task can perform the necessary eliminations prior to normalizing the row. On the basis of such a scheme, the sequential linear-equation solution is started *during* the loading process, thus reducing the size of the

pivot dependency graph that is to be normalized and eliminated after the Jacobian matrix has been loaded. With proper ordering, only the last few rows of the matrix will require triangularization and back-substitution after the model-evaluation phase. Further, the small normalization and elimination tasks are incorporated into larger device-evaluation tasks, thereby reducing the relative importance of the task handling and scheduling overhead.

For maximal efficiency from the pipelining it is necessary to order device evaluations such that the Jacobian coefficient matrix is loaded from top to bottom, i.e., in the order that the linear-equation solution proceeds. Thus, after the matrix has been re-ordered (after the first LU-decomposition), devices are assigned priorities according to the lowest row number that they update. Thus, for example, if device1 contributes to elements in rows 2, 4, 6 and 7, it is assigned a priority of 2. Now, if device2 contributes to elements in rows 1, 2 and 3, it is assigned a priority of 1. Then, the devices are placed in the task vector (described in Chapter 4) according to their respective priorities (device2 and then device1, in the preceding example). This scheme is designed to fill the early rows of the Jacobian matrix first, thereby exploiting the pipelining as efficiently as possible.

During model evaluation, each time a device evaluation is completed, the evaluating processor increments a count, *numCoefficientsDone*[*row*], for each of the rows affected by the device. Following this incrementing, the processor compares *numCoefficientsDone*[*row*] with the number of coefficients required for the particular row, *numCoefficientsReqd*[*row*]. If the two numbers are equal, implying that the row has been completely loaded, the processor performs the decomposition function, i.e., it eliminates all nonzeroes to the left of the pivot in the row, normalizes the row with respect to the pivot element and marks all the pivot's successor elements as being ready for elimination. Thus,

when the last processor has completed its model-evaluation task, the Jacobian matrix is completely triangularized as well.

In the parallel sparse linear-equation solver described in Chapter 5, the decomposition function for each pivot waits while its predecessor pivots are un-normalized. Since pivots are scheduled going from the top down in the PDG, the danger of deadlock is avoided implicitly since the scheduling scheme ensures that a pivot is not scheduled before its predecessors. However, with the ordering scheme described above, it is possible for a pivot to be scheduled before its predecessors. Although the ordering scheme reduces the risk of this happening, it does not completely eliminate the possibility of deadlock This is because while it does ensure that pivots near the top of the PDG are ready for normalization early, it does not ensure that pivots lower in the PDG are not also available for normalization early. Thus, it is possible that the ordering scheme may result in the early scheduling of pivots low in the PDG. If the number of such schedulings exceeds the number of processors available, it is evident that a condition of deadlock results, since all the available processors are waiting.

To avoid the possibility of deadlock described above, if a processor cannot normalize a pivot because one of the pivot's predecessors is not ready for normalization, the processor simply returns to further device evaluations (the row is already marked as being ready for normalization, since *numCoefficientsDone* = *numCoefficientsReqd*). Since the devices are ordered to maximize loading towards the top of the matrix, it is unlikely that there will be a large number of rows that require treatment as special cases. Thus, when a processor encounters a predecessor pivot that has not yet been normalized, the processor checks to see if the predecessor can be normalized before going on with further model evaluations.

In view of the low frequency of this occurrence, this scheme is viewed as being more efficient than other alternatives, such as a single processor performing the leftover linear-equation solution at the end of all the model evaluations, which would compromise the effect of pipelining significantly.

6.1.2 – Performance of Pipelined Algorithm

While pipelining model evaluation and matrix triangularization contributes to more efficient parallel linear-equation solution, it serves to reduce the parallelism during model evaluation by increasing interprocess communication time. In addition, the already small parallelism during linear-equation solution is also marginally compromised due to back-tracking when the pipelining results in the incorrect scheduling of a low-priority pivot for normalization. Thus, the ideal speedup for pipelined linearization and triangularization is lower than that for linearization alone, as is evident from Table 6.1.

The performance of the pipelined linearization-triangularization algorithm is compared against the performance of the simulator without pipelining in Table 6.1 above. It is evident that performance with pipelining is uniformly better than that without pipelining with a maximum improvement of about 16% for the RAM with 8 processors. Also, it can be seen that the effect of the pipelining is most noticeable as the size of the circuit and the number of processors increases.

6.2: Pipelined Linear-Equation Solution and Convergence Checking

For each Newton-Raphson iteration it is necessary to determine whether a stable solution has been reached, after the set of sparse linear equations have been solved. In Section 6.2.1, an algorithm for parallelizing convergence checking and pipelining it with the end of

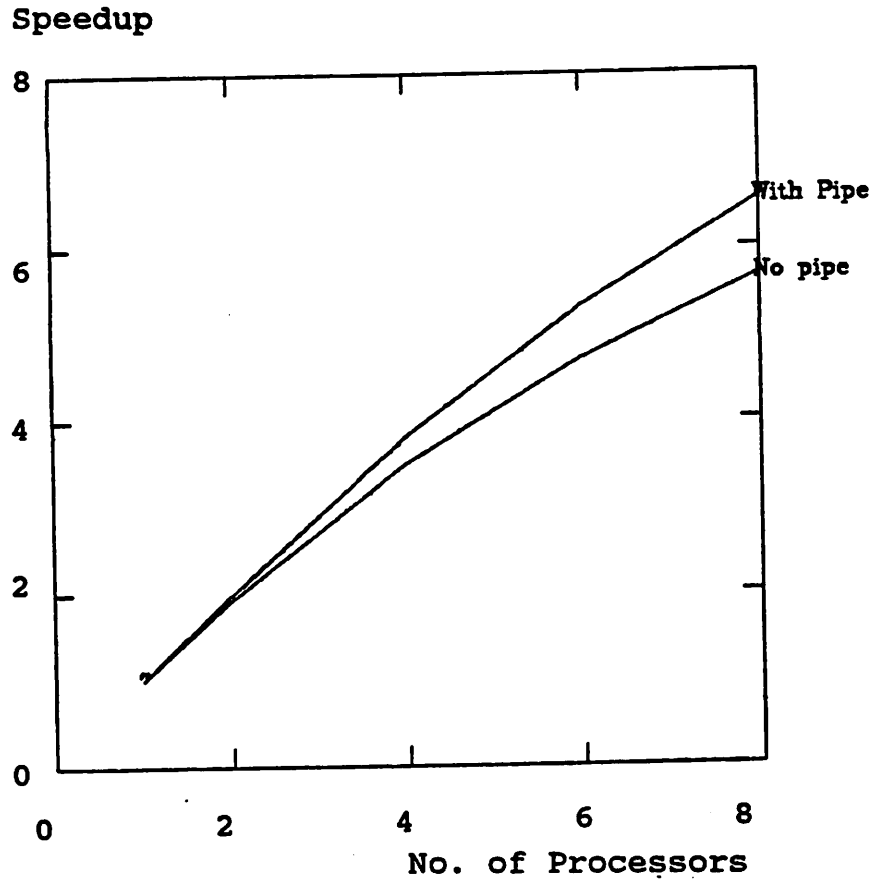

Figure 6.1: Pipelined and Unpipelined Speedup for CRAMB Benchmark

# of Procs	PLA		RAM		DAC	
	No pipe	Pipe	No pipe	Pipe	No pipe	Pipe
1	1.00	1.00	1.00	1.00	1.00	1.00
2	1.98	2.00	1.93	1.98	2.00	2.00
4	3.73	3.77	3.50	3.81	3.82	3.96
6	5.19	5.27	4.71	5.32	5.40	5.67
8	6.22	6.39	5.67	6.55	6.49	7.06

Table 6.1: Speedup due to Linearization and Triangulation
(Unpipelined and Pipelined)

the linear-equation solution phase that precedes it, is presented. Performance of the parallel simulator using this algorithm is evaluated in Section 6.2.2.

6.2.1 – Parallel Algorithm for Convergence Checking

A stable solution is defined as one where all node voltages and currents have settled within a certain tolerance limit. Thus, for example, the largest change in the node voltages between two consecutive iterations, *MaxDV*, is compared against pre-defined absolute and relative tolerance figures, *VAbsTol* and *VRelTol*, as given by the following equation:

$$MaxDV \leq VAbsTol + (VRelTol * AvgV) \qquad (6.1)$$

where *AvgV* is the average of all the node voltages. A similar equation is used to determine whether the currents flowing in the circuit have settled to within a weighted tolerance of the currents at the previous iteration.

For a uniprocessor direct-method solver, the algorithm used to determine whether the solution has converged is as follows:

Convergence Checking:

```
Assume convergence.
/* Determine maximum delta V, i.e. maximum change in node voltage */
MaxDV = 0.0;
foreach ( row i ) {
        MaxDV = MAX( MaxDV, dabs(rhs[i]));
}
/* Voltage limiting, if maximum delta V exceeds nralpha */
if ( MaxDV > nralpha ) {
        temp = nralpha / MaxDV;
        foreach ( row i ) {
                rhs[i] = rhs[i] * temp;
        }
}
/* Determine average node voltage */
AvgV = 0.0;
foreach ( row i ) {
```

```
        nodevolt[i] = nodevolt[i] - rhs[i];
        AvgV = AvgV + dabs(nodevolt[i]);
}
AvgV /= number of rows;
/* If max delta V exceeds threshold of tolerance, not converged */
if ( MaxDV > (VAbsTol + (VRelTol * AvgV)) ) {
        not converged;
/* Else, examine currents */
} else {
        Evaluate models and Load the right-hand side vector;
        foreach ( row i ) {
                dtemp = alpha0 * charge[i];
                rhs[i] = dtemp + oldq[i] + current[i];
                isum = dabs(dtemp) + dabs(oldq[i]) + dabs(current[i]);
                /* If current exceeds threshold, not converged */
                if ( dabs(rhs[i]) > ((IRelTol * isum) + IAbsTol) ) {
                        not converged;
                }
        }
}
```

Given the serial algorithm above, an algorithm for parallelizing convergence checking, as well as for pipelining linear-equation solution with convergence checking has been developed. This procedure is presented below.

It is evident from the serial convergence-checking algorithm that until the maximum change in node voltages, *MaxDV*, has been determined, none of the other convergence checking operations can be performed, since any voltage limiting depends on whether *nralpha* is exceeded. Thus, determination of *MaxDV* presents a bottleneck to pipelining linear-equation solution and convergence checking. However, the determination of *MaxDV* itself can be easily pipelined with the back-substitution phase of linear-equation solution, since it depends only on the value of *rhs*[*i*], for a given row *i*. Thus, for pipelined convergence checking, *MaxDV* is updated as each row is solved for during back-substitution.

If *MaxDV* exceeds *nralpha*, it is necessary to scale the change in the node voltages. Since this operation must be performed on each row of the matrix, it has a parallel speedup

potential equal to the dimension of the matrix, ignoring scheduling and task handling overheads. In addition, since these operations are performed on the *rhs* array, it is well suited to vector processing.

Although the next step, which is the determination of the average node voltage, can be pipelined with voltage limiting to eliminate the synchronization point between the two steps, voltage limiting is not performed frequently enough to justify this pipelining, and the two phases are parallelized individually and independently. Determination of average node voltage is done as follows: each processor updates node voltages for all the nodes it has been assigned, maintains a local sum of all its assigned node voltages, and then, synchronizing through the use of a global lock, adds its local sum into a global sum for the circuit's average node voltage.

After another synchronization bottleneck, i.e., the determination of whether the node voltages have converged, convergence is checked for the currents. The model evaluation for the loading of the right-hand side vector is performed in parallel using the same algorithm as is described in Chapter 4 for loading of the Jacobian coefficient matrix.

Once the right-hand side vector has been loaded, the serial algorithm determines whether the solution has converged by examining the current for each node in the circuit. Using the parallel model of each processor working on a dynamically assigned set of nodes, each processor performs the same current convergence checking operation, but only for those rows (i.e. those nodes) that it has been assigned. If at any stage a processor determines that convergence has not been achieved, it can update the global flag for convergence and suspend further convergence checking.

Note that the serialization between convergence checking for voltages and currents is unnecessary, and that parallel checking of both sets of variables is possible if the degree of parallelism available is small, or if the overhead associated with convergence checking is large. Indeed, since current convergence checking takes a longer time, due to the model evaluation phase, this phase is started at the same time as voltage convergence checking, and processors constantly monitor the global convergence flag to cut short their individual checking efforts in the event of non-convergence among one of the other processors' assigned node voltages or currents.

6.2.2 – Performance Evaluation

The algorithms described in section 6.2.1 have been implemented and simulation results from this implementation are presented in Table 6.2.

The significant improvement due to pipelining and parallel convergence checking is due to the model evaluation for the loading of the right-hand side vector, which is performed in parallel using the same algorithm as is used for linearization. As with parallel linearization, significant reduction in simulation time results from parallel loading of the right-hand side vector, as is evident from Table 6.2.

Figure 6.3 shows a profile of the number of processors busy during a single iteration of the Newton-Raphson loop for the Cramb benchmark circuit, with model evaluation and matrix triangulation being performed independently and sequentially. For this profiling, 12 processors were available on the Sequent Balance, of which 11 were used. It is evident that after the model evaluation, which keeps all 11 processors busy, matrix triangulation and back-substitution are unable to utilize all the processors consistently. The last pulse in

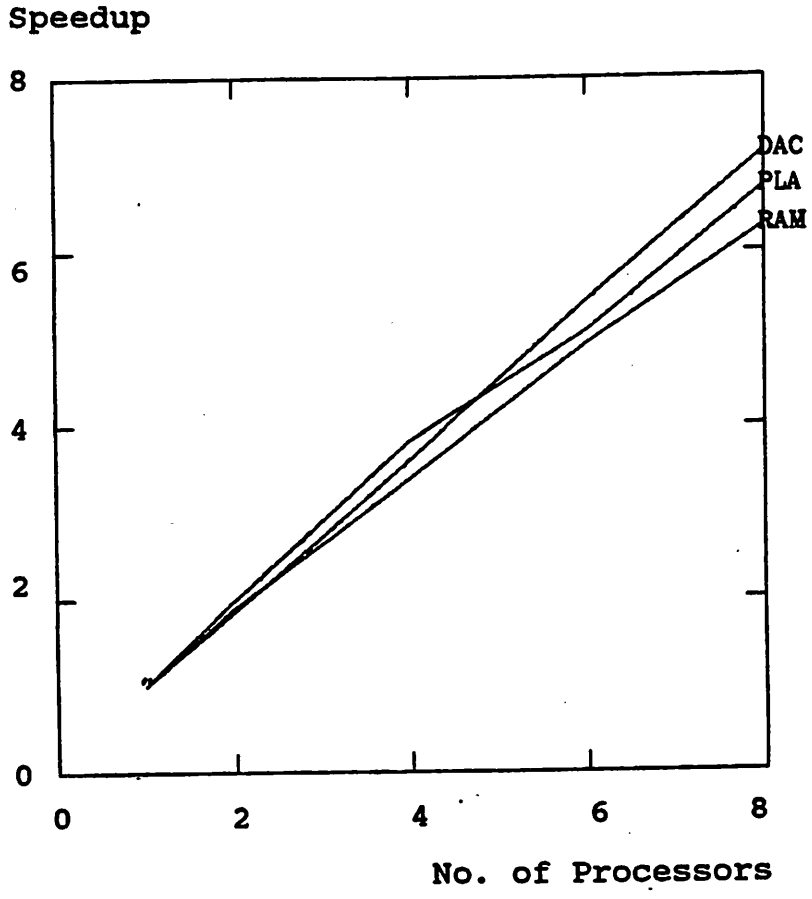

Figure 6.2: Speedup for Parallel RHS Loading

# of Procs	PLA	RAM	DAC
1	1.00	1.00	1.00
2	1.99	1.90	1.86
4	3.83	3.40	3.61
6	5.08	4.93	5.43
8	6.73	6.28	7.14

Table 6.2: Speedup due to Parallel Loading of the Right-hand Side Vector

the graph displays the high processor utilization during convergence checking.

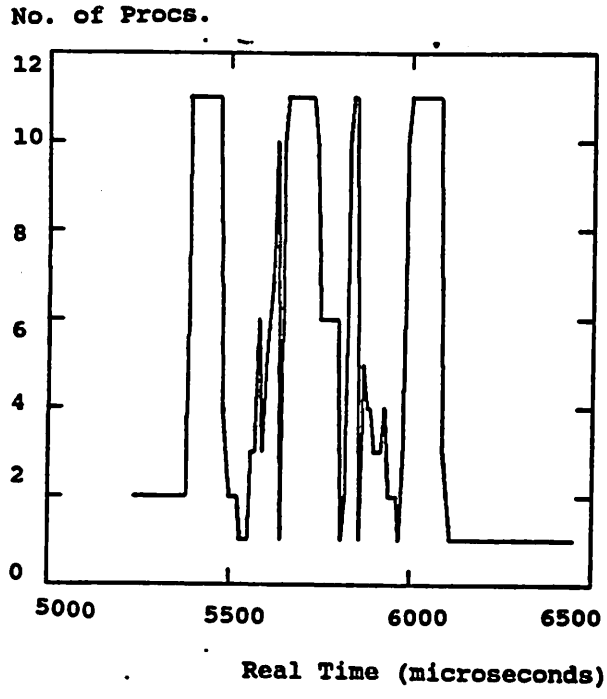

Figure 6.3: Processor Activity Profile during Sequential Newton-Raphson iteration (Cramb benchmark, using 11 processors)

In Figure 6.4, the processor activity profile is shown *with* pipelining between the model evaluation and matrix-triangulation phases. As in Figure 6.3, 11 of a 12-processor Sequent Balance were used for this profiling. It is evident that the pipelining leads to an increased model-evaluation time, but a significantly reduced triangulation time. Thus, the total time between the beginning of the model evaluation and the end of the convergence checking falls from 690ms without pipelining to 570ms with pipelining. This yields a speed improvement of about 20%, which is slightly compromised by the additional overhead for setting up data structures to facilitate pipelining.

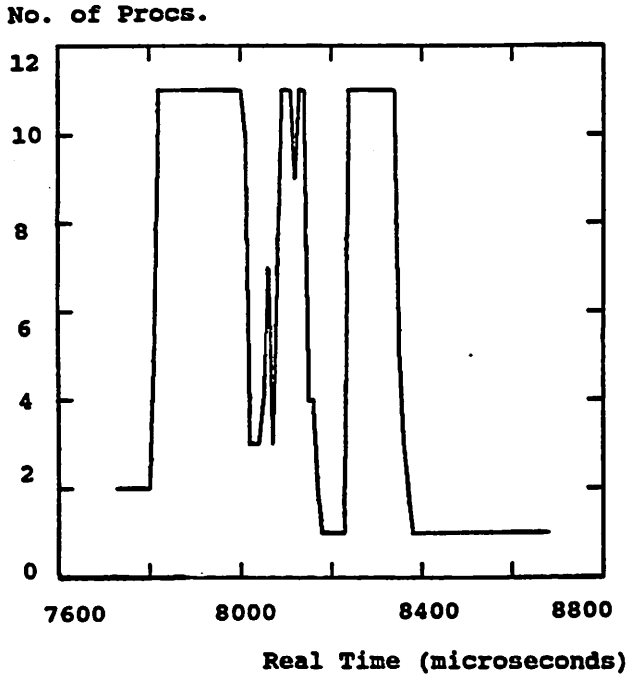

Figure 6.4: Processor Activity Profile during Pipelined Newton-Raphson iteration (Cramb benchmark, using 11 processors)

6.3: Effect of inter-phase synchronization on overall parallel performance

Table 6.3 and Figure 6.5 show the improvement in the parallel direct-method solver due to the combined effect of pipelining and exploitation of parallelism at each of the important stages of the simulation algorithm. It is evident that the effect of the efficient parallelization of model evaluation results in higher speedups for the PLA and DAC circuits, for which model evaluation takes 72% and 71% of the total simulation time. Also interesting is the fact that overall speedup for the DAC benchmark circuit is higher than for the PLA circuit, which has a higher fraction of model evaluation time. This result reflects the importance of efficient pipelining of model evaluation with linear-equation solution.

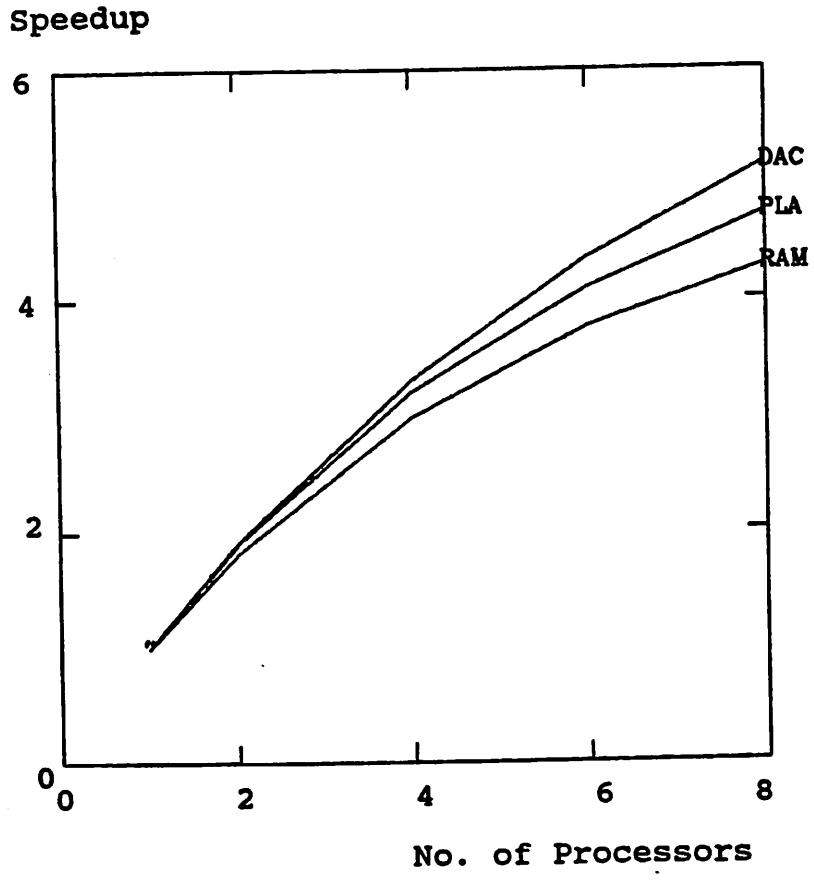

Figure 6.5: Overall Speedup for Direct-Method Simulation (DAC Benchmark Circuit)

# of Procs	PLA	RAM	DAC
1	1.00	1.00	1.00
2	1.88	1.80	1.89
4	3.20	2.98	3.30
6	4.10	3.76	4.36
8	4.75	4.29	5.18

Table 6.3: Overall Direct-method Simulation Speedup due to Pipelining

As is mentioned in Chapter 4, the parallel direct-method solver developed in this study is part of an ITA relaxation simulator, PSPLICE3. The parallel performance of PSPLICE3 on the 8-processor Sequent Balance achieved a maximum speedup of a factor of about 3.25 by merely scheduling different subcircuits in parallel [43]. This low speedup is obtained because only a few subcircuits are active at a given time during the simulation (a feature characteristic of predominantly digital circuits) and poor balancing of the load between the different processors. Sub-division of a subcircuit evaluation task results in improved speedup figures for the parallel relaxation simulator as is shown in Chapter 4.

The improvements in parallelizing the direct-method solver, detailed in Chapter 5 and in this chapter, lead to corresponding improvements in the speed performance of PSPLICE3, the parallel relaxation-direct simulator. Speedup figures for the three benchmark circuits after the implementation of parallel linear-equation solution, the pipelining of model evaluation and linear-equation solution and the parallelization and pipelining of convergence checking are shown in Table 6.4 and Figure 6.6. It is evident that a parallel direct-method solver significantly improves the performance of a parallel relaxation-direct simulator, primarily because the elemental tasks of the former have a far finer granularity and are not as dependent on the connectivity of the circuit being simulated.

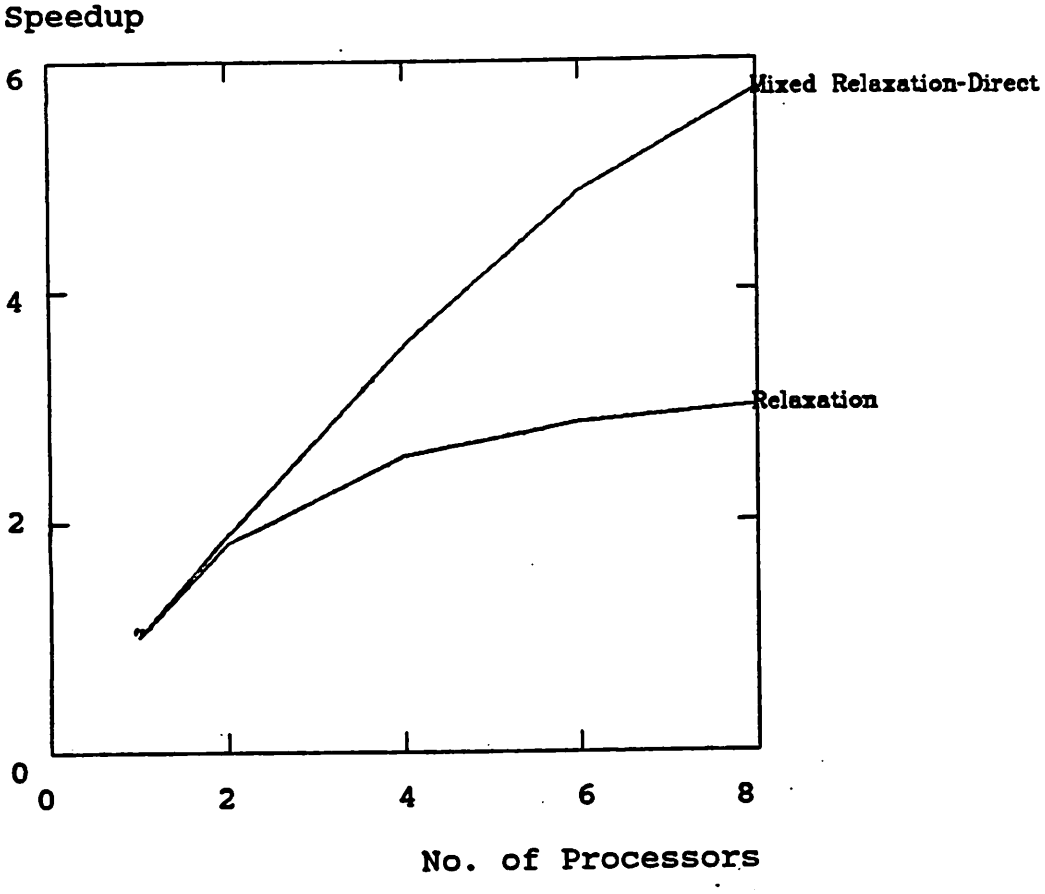

Figure 6.6: Relaxation and Mixed Direct-Relaxation Speedup for RAM

# of Procs	PLA		RAM		DAC	
	(a)	(b)	(a)	(b)	(a)	(b)
1	1.00	1.00	1.00	1.00	1.00	1.00
2	1.85	1.84	1.81	1.79	1.81	1.88
4	3.16	3.27	2.47	2.93	2.56	3.52
6	3.59	4.11	2.62	3.61	2.86	4.86
8	3.22	5.11	2.71	3.89	2.99	5.74

Table 6.4: Relaxation Simulator Speedup

(a) Parallel Relaxation, and (b) Parallel Mixed Relaxation-Direct methods"

CHAPTER 7

CONCLUSIONS

A number of issues pertaining to parallel processing in circuit simulation are presented in this dissertation. This work establishes that node-based relaxation circuit simulation is well-suited to parallel processing on up to 100 processors. It is observed that model evaluation in parallel direct-method circuit simulation can be accelerated through the use of parallel processing; however, linear-equation solution and the synchronization points in the standard direct-method simulation algorithm prove to hinder efficient parallel simulation. A new pipelined algorithm is presented and is shown to reduce the time for parallel direct-method circuit simulation.

From the study and extension of MSPLICE, a node-based relaxation circuit simulator using up to 100 processors of the omega-connected BBN Butterfly Parallel Processor, it is observed that efficient parallel processing in a shared-memory environment requires proper distribution of data: multiple copies of shared static data and an equitable distribution of shared dynamic data serve to keep contention for the interconnection network and memory low. It is also seen that while both dynamic and static scheduling perform well on small parallel-processor systems, single-queue dynamic scheduling is most efficient as the number of processors increases, simultaneously addressing the issues of load balancing and queue selection. Using MSPLICE2, up to 78% of ideal speedup performance predicted by profiling for a 222-node, 517-MOSFET transistor counter benchmark circuit is achieved. With floating-point hardware acceleration, simulations using MSPLICE2 on a 32-processor Butterfly run about as fast as MSPLICE simulations on a VAX 8800.

Parallel direct-method circuit simulation has been studied using the subcircuit solver within the PSPLICE3 mixed direct-relaxation simulator, implemented on up to 8 processors of the Sequent Balance B8000. The new program is referred to as PDSPLICE3. The two most time-consuming phases of direct-method simulation, model evaluation and linear-equation solution, are studied in detail. It is seen that model evaluation is well-suited to parallel processing and that model-evaluation time decreases almost linearly with increasing numbers of processors. However, linear-equation solution is observed to have low parallelism potential due to matrix sparsity, which results in small task sizes, and matrix fill-ins which impose a sequential schedule of operations towards the end of the matrix triangulation and at the beginning of the back substitution. Further, the serial synchronization point between the parallelized model evaluation and linear-equation solution phases further reduces the efficiency of the parallel direct-method simulator.

A new pipelined parallel direct-method circuit-simulation algorithm is presented that simultaneously addresses the problems of poor linear-equation solution parallelization and the inter-phase bottleneck. The combination of model-evaluation and linear-equation solution tasks serves to reduce the relative overhead associated with each of the individual tasks. Further, pipelined model evaluation and linear-equation solution, where triangulation begins at the top of the Jacobian matrix while the bottom of the matrix is still being evaluated, ameliorates the effect of the inter-phase bottleneck. The resulting pipelined, parallel direct-method solver, PDSPLICE3, significantly reduces circuit-simulation runtimes. For a 155-node, 416-MOSFET transistor digital-to-analog circuit, simulation speedup using parallel direct methods is 5.18 on an 8-processor Sequent, while a speedup of 5.74 is achieved using parallel mixed direct-relaxation techniques.

There are a number of areas where further research is required:

(a) schemes of matrix ordering, such as representation in a Bordered Block Diagonal Form or using nested dissection, that will make linear-equation solution more parallel -- this will require heuristic schemes to partition the matrix;

(b) improved pipelining between model evaluation and linear-equation solution to further reduce the effect of the synchronization point between the two phases;

(c) investigation of matrix-assembly techniques, such as Sparse Tableau Analysis, to determine whether memory usage can be traded off against speed -- although a larger matrix may increase the parallelism due to sparsity, partitioning of the matrix will probably be more difficult;

(d) extension of the results presented in this dissertation to larger systems (medium-scale, such as the BBN Butterfly, and large-scale, such as the TMC Connection Machine [82]);

(e) parallelization of sequential phases in direct-method simulation that become more important as the more parallel phases take less time due to parallel processing.

APPENDIX A

The Source Listing of the Program PDSPLICE3 is available at the following address:

Software Distribution Office,

Industrial Liaison Program,

Department of Electrical Engineering and Computer Sciences,

University of California at Berkeley,

Berkeley, CA 94720.

APPENDIX B

The Source Listing of the Program MSPLICE1 is available at the following address:

Software Distribution Office,

Industrial Liaison Program,

Department of Electrical Engineering and Computer Sciences,

University of California at Berkeley,

Berkeley, CA 94720.

APPENDIX C

The Source Listing of the Program MSPLICE2 is available at the following address:

Software Distribution Office,

Industrial Liaison Program,

Department of Electrical Engineering and Computer Sciences,

University of California at Berkeley,

Berkeley, CA 94720.

References

1. L.W. Nagel, "ADVICE for Circuit Simulation," *presented at IEEE International Symposium on Circuits and Systems*, Houston, TX, May 1980.

2. W.T. Weeks and A.J. Jimenez, G.W. Malhoney, D. Mehta, H. Qassemzadeh, T.R. Scott, "Algorithms for ASTAP -- A Network Analysis Program," *IEEE Trans. on Circuit Theory*, vol. CT-20, pp. 628-634, Nov. 1973.

3. P. Yang and I.N. Hajj, T.N. Trick, "SLATE: A Circuit Simulation Program with Latency Exploitation and Node Tearing," *Proc. IEEE International Conference on Circuits and Computers*, vol. 1, pp. 353-355, New York, NY, Oct. 1980.

4. L.W. Nagel, "SPICE2: A Computer Program to Simulate Semiconductor Circuits," *Memo No. ERL-M520*, University of California, Berkeley, May 1975.

5. K. Sakallah and S.W. Director, "An Activity-Directed Circuit Simulation Algorithm," *Proc. IEEE International Conference on Circuits and Computers*, pp. 1032-1035, New York, NY, Oct. 1980.

6. A.R. Newton, "The Simulation of Large-Scale Integrated Circuits," *Memo No. UCB/ERL M78/52*, University of California, Berkeley, July 1978.

7. R. Saleh and J.E. Kleckner, A.R. Newton, "Iterated Timing Analysis in SPLICE1," *Proc. IEEE International Conference on Computer-Aided Design*, pp. 139-140, Santa Clara, CA, Sep. 1983.

8. J.E. Kleckner, "Advanced Mixed-Mode Simulation Techniques," *UCB/ERL Memo No. M84/48*, University of California, Berkeley, May 1984.

9. E. Lelarasmee, "The Waveform Relaxation Method for Time Domain Analysis of Large Scale Integrated Circuits: Theory and Applications," *Memo No. UCB/ERL M82/40*, University of California, Berkeley, May 1982.

10. J.K. White, "The Multirate Integration Properties of Waveform Relaxation, with Applications to Circuit Simulation and Parallel Computation," *Memo No. UCB/ERL 85/90*,

University of California, Berkeley, Nov. 1985.

11. R. Lucas and T. Blank, J. Tiemann, "A Parallel Solution Method for Large Sparse System of Equations," *Proc. IEEE International Conference on Computer-Aided Design* , pp. 178-181, Santa Clara, CA, Nov. 1986.

12. A. Vladimirescu and D. Weiss, M. Katevenis, Z. Bronstein, A. Kfir, K. Danuwidjaja, K.C. Ng, N. Jain, S. Lass, "A Vector Hardware Accelerator with Circuit Simulation Emphasis," *Proc. 24th ACM/IEEE Design Automation Conference*, pp. 89-94, Miami, FL, June 1987.

13. G. Bischoff and S. Greenberg, "CAYENNE: A Parallel Implementation of the Circuit Simulator SPICE," *Proc. IEEE International Conference on Computer-Aided Design* , pp. 182-185, Santa Clara, CA, Nov. 1986.

14. P. Cox and R. Burch, B. Epler, "Circuit Partitioning for Parallel Processing," *Proc. IEEE International Conference on Computer-Aided Design* , pp. 186-189, Santa Clara, CA, Nov. 1986.

15. S. Mattisson, "CONCISE: A Concurrent Circuit Simulation Program," *Memo LUTEDX/(TETE-1003)/1-116/(1986)*, Department of Applied Electronics, University of Lund, Lund, Sweden, Aug. 1986.

16. P. Sadayappan and V. Visvanathan, "Circuit Simulation on a Multiprocessor," *Proc. Custom Integrated Circuits Conference*, pp. 124-128, Portland, OR, May 4-7, 1987.

17. J. White and N. Weiner, "Parallelizing Circuit Simulation - A Combined Algorithmic and Specialized Hardware Approach," *Proc. IEEE International Conference on Computers and Design*, pp. 438-441, New York, NY, Oct. 1985.

18. D.M. Lewis, "A High Performance Hardware Accelerator for Circuit Level Simulation of VLSI Circuits," *Proc. IEEE International Conference on Computer-Aided Design* , pp. 386-389, Santa Clara, CA, Nov. 1986.

19. E. Cohen, "Performance Limits of Integrated Circuit Simulation on a Dedicated Mini-

computer System," *Memo No. UCB/ERL M81/29*, University of California, Berkeley, May 1981.

20. J.L. Burns, "Empirical Mosfet Models for Circuit Simulation," *Memo No. UCB/ERL M84/43*, University of California, Berkeley, May 1984.

21. T. Shima and T. Sugawara, S. Moriyama, H. Yamada, "Three-Dimensional Table Look-Up MOSFET Model for Precise Circuit Simulation," *IEEE Journal of Solid-State Circuits*, vol. SC-17, pp. 449-454, June 1982.

22. J. Barby and J. Vlach, K. Singhal, "Optimized Polynomial Splines for FET Models," *Proc. IEEE International Symposium on Circuits and Systems*, pp. 1159-1162, Montreal, Canada, May 1984.

23. R.S. Gyurcsik, "An Attached Processor for MOS-transistor Model Evaluation," *Memo No. UCB/ERL M86/82*, University of California, Berkeley, Oct. 1986.

24. A. Vladimirescu, "LSI Circuit Simulation on Vector Computers," *Memo No. UCB/ERL M82/75*, University of California, Berkeley, Oct. 1982.

25. D.A. Calahan, "Vector processors - Models and applications," *IEEE Trans. on Circuits and Systems*, vol. CAS-26, pp. 715-726, 1979.

26. CRAY Research, Inc., "CRAY-1 Computer Systems Hardware Description Manual," *Publication Number 2240004*, Mendota Heights, Minnesota, 1976.

27. A. Vladimirescu, "LSI Circuit Simulation on Attached Array Processors," *Memo No. UCB/ERL M84/6*, University of California, Berkeley, Jan. 1984.

28. L.J. Shanbeck and R.S. Norin, "QSPICE: An Application of Array Processors to CAD Simulation," *Proc. IEEE International Conference on Computer-Aided Design*, pp. 247-249, Santa Clara, CA, Sep. 1983.

29. A.E. Charlesworth, "An Approach to Scientific Array Processing: The Design of the SP-120B/FPS-164 Family"," *Computer Magazine*, vol. 14, no. 9, Sep. 1981.

30. A. Vladimirescu and K. Zhang, A.R. Newton, D.O. Pederson, A. Sangiovanni-Vincentelli, *SPICE Version 2G User's Guide*, University of California, Berkeley, Aug. 1981.

31. CRAY Research, Inc., "The CRAY X-MP Series of Computers," *Publication MP-0001*, Mendota Heights, MN, 1982.

32. S. Nagashima and Y. Inagami, T. Oduka, S. Kanabe, "Design Considerations for a High-Speed Vector Processor," *IEEE International Conference on Computer Design*, pp. 238-243, Port Chester, NY, Oct. 8-11, 1984.

33. Control Data Corporation, "CDC Cyber 200 Model 203 Computer System Hardware Reference Manual," *Publication no. 60256010*, St.Paul, MN, May 1980.

34. F. Yamamoto and S. Takahashi, "Vectorized LU Decomposition Algorithms for Large-Scale Circuit Simulation," *IEEE Trans. on Computer-Aided Design*, vol. CAD-4, no. 3, pp. 232-239, July 1985.

35. L.C. Higbie, "Supercomputer Architecture," *Computer*, pp. 48-58, Dec. 1973.

36. B. Greer, "Converting SPICE to Vector Code," *VLSI Systems Design*, vol. VII, no. 1, pp. 30-35, Jan. 1986.

37. Bolt, Beranek and Newman, "Butterfly Parallel Processor Overview, Version 1," *Users' Guide*, May 1985.

38. Intel Scientific Computers, "iPSC: The First Family of Concurrent Supercomputers," *Product Announcement*, Portland, Oregon., 1985.

39. Sequent Computer Systems, Inc., "Balance 8000 Guide to Parallel Programming," *Users' Guide*, July 1985.

40. Alliant Computer Systems Corporation, "FX/Series Architecture Manual," *Users' Guide*, Jan. 1986.

41. ELXSI, "ELXSI System 6400 System Introduction," *Users' Guide*, 1984.

42. A.R. Newton and A.L. Sangiovanni-Vincentelli, "Relaxation-based Circuit Simulation," *IEEE Trans. on ED*, vol. ED-30, no. 9, pp. 1184-1207, Sep. 1983.

43. R.A. Saleh, "Nonlinear Relaxation Algorithms for Circuit Simulation," *Memo No. UCB/ERL M87/21*, University of California, Berkeley, 15 April, 1987.

44. D.M. Webber and A. Sangiovanni-Vincentelli, "Circuit Simulation on the Connection Machine," *24th ACM/IEEE Design Automation Conference 1987 Proceedings*, pp. 108-113, Miami, FL, June 1987.

45. O. Wing and J.W. Huang, "A Computation Model of Parallel Solution of Linear Equations," *IEEE Trans. on Computers*, vol. C-29, no. 7, pp. 632-638, July 1980.

46. A.L. Sangiovanni-Vincentelli, "Circuit Simulation," in *Computer Design Aids for VLSI Circuits*, ed. P. Antognetti, D.O. Pederson, H. De Man, pp. 19-113, Sijthoff & Noordhoff, 1981.

47. C.A. Desoer and E.S. Kuh, *Basic Circuit Theory*, McGraw-Hill, 1969.

48. G.D. Hachtel and R.K. Brayton, F.G. Gustavson, "The Sparse Tableau Approach to Network Analysis and Design," *IEEE Trans. on Circuit Theory*, vol. CT-18, pp. 101-113, Jan. 1971.

49. C.W. Ho and A.E.Ruehli, P.A. Brennan, "The Modified Nodal Approach to Network Analysis," *IEEE Trans. on Circuits and Systems*, vol. CAS-22, pp. 504-509, June 1975.

50. P. Yang, "An Investigation of Ordering, Tearing and Latency Algorithms for the Time-Domain Simulation of Large Circuits," *Report R-891, Coordinated Science Lab.*, University of Illinois, Urbana, Aug. 1980.

51. C.W. Gear, *Numerical Initial Value Problems for Ordinary Differential Equations*, Prentice Hall, 1974.

52. J.L. Burns and A.R. Newton, D.O. Pederson, "Active Device Table Look-Up Models for Circuit Simulation," *Proc. IEEE International Symposium on Circuits and Systems*, pp. 250-253, Newport Beach, CA, May 1983.

53. W.J. McCalla, "Computer-Aided Circuit Simulation Techniques," *Pre-publication manuscript*, Kluwer Academic Publishers.

54. H.M. Markowitz, "The Elimination Form of the Inverse and Its Application to Linear Programming," *Management Science*, vol. 3, pp. 255-269, April 1957.

55. A. Vladimirescu, "Accuracy Enhancement in SPICE2 Using Sparse Matrix Pivoting Techniques," *EECS 290H Class Project*, University of California, Berkeley, Winter 1978.

56. H.F. Ko, "A Special-Purpose Architecture and Parallel Algorithms on a Multiprocessor System for the Solution of Large Scale Linear Systems of Equations," *Ph.D. dissertation*, University of California, Berkeley, CA, Aug. 1986.

57. A. Sangiovanni-Vincentelli and L.K. Chen, L.O. Chua, "A New Tearing Approach -- Node-Tearing Nodal Analysis," *Proc. IEEE International Symposium on Circuits and Systems*, pp. 143-147, 1977.

58. H.J. De Man, "Computer-Aided Design for Integrated Circuits: Trying to Bridge the Gap," *IEEE Journal of Solid-State Circuits*, vol. SC-14, pp. 613-621, June 1979.

59. R.S. Gyurcsik, "BIASC: A Circuit Simulation Program for the IBM PC," *Wescon/85 Professional Program Session Record 32/4*, San Francisco, CA, Nov. 1985.

60. H. Uno and H. Kinoshita, S. Kumagai, I. Shirakawa, S. Kodama, "A Parallel Implementation of MOS Digital Circuit Simulation," *Proc. IEEE International Conference on Computer-Aided Design*, pp. 2-4, Santa Clara, CA, Nov. 1985.

61. R. Thomas, "Using the Butterfly to Solve Simultaneous Linear Equations," *BBN Labs Memorandum*, Cambridge, MA, March 1985.

62. J.T. Deutsch, "Algorithms and Architecture for Multiprocessor-based Circuit Simulation," *Memo No. UCB/ERL M85/39*, University of Calfornia, Berkeley, May 1985.

63. J.T. Deutsch and A.R. Newton, "MSPLICE: A Multiprocessor Based Circuit Simulator," *Proc. International Conference on Parallel Processing*, pp. 207-214, May 1984.

64. J.T. Deutsch and A.R. Newton, "A Multiprocessor Implementation of Relaxation-Based Electrical Circuit Simulation," *21st ACM/IEEE Design Automation Conference Proceedings*, pp. 350-357, Albuquerque, NM, June 1984.

65. G.K. Jacob and A.R. Newton, D.O. Pederson, "An Empirical Analysis of the Performance of a Multiprocessor-based Circuit Simulator," *23rd ACM/IEEE Design Automation Conference Proceedings*, pp. 588-593, Las Vegas, Nevada, June 1986.

66. R. Rettberg and F. Heart, B. Mann, J. Goodhue, *The Chrysalis Operating System Manual*, Bolt, Beranek and Newman, Cambridge, MA, June 1983.

67. Bolt, Beranek and Newman, *Chrysalis Programmers' Manual, Version 2.2*, Cambridge, MA, June 20, 1985.

68. R.A. Saleh and A.R. Newton, "An Event-Driven Relaxation-Based Multirate Integration Scheme for Circuit Simulation," *Proc. IEEE International Symposium on Circuits and Systems*, pp. 600-603, Philadelphia, PA, May 1987.

69. B.R. Chawla and H.K. Gummel, P. Kozak, "MOTIS - An MOS Timing Simulator," *IEEE Trans. on Circuits and Systems*, vol. CAS-22, pp. 901-909, Dec. 1975.

70. A.R. Newton, "Pidgin 'C': The only way to fly," *ERL Memo in preparation*, University of California, Berkeley, CA.

71. D.H. Lawrie, "Access and Alignment of Data in an Array Processor," *IEEE Trans. on Computers*, vol. C-24, no. 12, pp. 1145-1155., Dec. 1975.

72. E.O. Brigham, *The Fast Fourier Transform*, Prentice-Hall, Englewood Cliffs, NJ, 1974.

73. C.A.R. Hoare, *Communicating Sequential Processes*, Prentice-Hall International, Englewood Cliffs, N.J., 1985.

74. G. Fielland and D. Rogers, "32-bit Computer System Shares Load Equally Among up to 12 Processors," *Electronic Design*, pp. 153-168, Sep. 1984.

75. J.R. Goodman, "Using Cache Memory to Reduce Processor-Memory Traffic," *IEEE Computer Architecture Symposium Proc.*, pp. 124-131, 1983.

76. R. Katz and S. Eggers, D. WOod, C.L. Perkins, R. Sheldon, "Implementing a Cache Consistency Protocol," *Proc. 12th Annual International Symposium on Computer Architecure*, vol. 13, no. 3, pp. 276-283, Boston, MA, June 1985.

77. G.M. Amdahl, "Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities," *AFIPS Conf. Proc.*, vol. 30, pp. 483-485, 1967.

78. D.J. Kuck and R.H. Kuhn, D.A. Padua, B. Leasure, M. Wolfe, "Dependence graphs and compiler optimizations," *8th Annual ACM Symp. on Principles of Programming Languages*, pp. 207-218, Williamsburg, VA, Jan. 26-28, 1981.

79. G.K. Jacob and A.R. Newton, D.O. Pederson, "Direct-Method Circuit Simulation using Multiprocessors," *Proc. IEEE International Symposium on Circuits and Systems* , vol. 1, pp. 170-173, San Jose, CA, May 1986.

80. G.K. Jacob and A.R. Newton, D.O. Pederson, "Parallel Linear-Equation Solution in Direct-Method Circuit Simulators," *Proc. IEEE International Symposium on Circuits and Systems*, vol. 3, pp. 1056-1059, Philadelphia, PA, May 4-7, 1987.

81. K.S. Kundert, "Sparse Matrix Techniques and their Application to Circuit Simulation," in *Circuit Analysis, Simulation and Design*, ed. A.E.Ruehli, pp. 281-324, North-Holland Co., Cambridge, MA, 1986.

82. W.D. Hillis, "The Connection Machine," *Ph.D. Thesis, Dept. of EECS, MIT*, Cambridge, MA, May, 1985.