Circuit Simulation Algorithms for Massively Parallel Processors

by

Luís Miguel Silveira

Engenheiro Electrotécnico e de Computadores,
Instituto Superior Técnico, Lisboa, Portugal (1986)

Mestre em Engenharia Electrotécnica e de Computadores,
Instituto Superior Técnico, Lisboa, Portugal (1989)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 1990

© Massachusetts Institute of Technology 1990

Signature of Author_____
Department of Electrical Engineering and Computer Science
October 20, 1995

Certified by_____
Jacob K. White
Thesis Supervisor

Accepted by_____
Arthur C. Smith
Chairman, Departmental Committee on Graduate Students

Circuit Simulation Algorithms for Massively Parallel Processors

by

Luís Miguel Silveira

Submitted to the Department of Electrical Engineering and Computer Science
on October 20, 1995, in partial fulfillment of the
requirements for the degree of
Master of Science

Abstract

The cost in time and capital to fabricate an integrated circuit has made circuit simulation an indispensable, though computationally very expensive, tool. Parallel processing has been considered as a way to reduce the turnaround time for large circuit simulation, and the recently available massively parallel computers are particularly attractive for accelerating the simulation of certain classes of circuits. This thesis examines the problem of efficiently exploiting the computational resources of massively parallel computers for which new algorithms and techniques must be devised. Throughout this study it became evident that problems or circuits possessing certain intrinsic features are inherently more appropriate for implementation on those types of machines. The so-called "vision circuits", circuits implementing early vision tasks, form a class of circuits that, for the most part cannot be simulated with traditional programs due to their large sizes and the degree of accuracy required. Moreover their extreme regularity makes them obvious candidates for simulation on massively parallel computer architectures. In this thesis, their functionality is studied by casting their behavior into the framework of numerical circuit simulation. Specific integration formulas are examined as viable alternatives for solving the problem. A methodology is developed that, when implemented on a commercial massively parallel processor, allows over 1 Gigaflop performance and makes it possible for this type of circuits to be simulated and their behavior tested before fabrication.

Thesis Supervisor: Jacob K. White

Associate Professor of Electrical Engineering and Computer Science

To my parents

Maria Luísa d' Ávila Pinto da Silveira
Fernando Augusto Teixeira da Silveira

to whom I owe everything. For their continuing love and support.

Acknowledgments

The work described in this thesis would not have been possible without the guidance and expertise of my thesis adviser and mentor, Jacob K. White. I am forever grateful for him having put me on the track of numerical circuit simulation and providing the knowledge, the help and the motivation to pursue that study.

Very special thanks are in order for Andrew Lumsdaine whose knowledge and interest on the problem at hand have motivated and lead me through the duration of this research, and without whom the implementation work would never have been achieved. His extensive editing and correcting of earlier drafts of this thesis have strongly contributed to improving its structure and contents.

Several other members of my research group have contributed directly or indirectly to the work described in this thesis. Keith Nabors with his friendshipness has been a formidable source of encouragement. Ricardo Telichevesky explained many questions related to parallel computer architectures and Mark Reichelt always provided helpful hints about all questions we ever discussed.

In addition to the mentioned, I would like to thank all the members of my research group for their continuing support and their effort over the past two years. Other MIT related people, mainly some Portuguese colleagues in other departments, deserve some acknowledgment for having made my stay a lot more bearable. Horácio Neto, a graduate student in my research group in Portugal was behind most of the work and results related to exponential fitting.

Thinking Machines Corporation and especially Rolf Fiebrich and his group also deserve acknowledgment for letting me use their resources and for providing the necessary advice and expertise during the course of this work.

Foremost, I am forever indebted to my parents, my brother and my wife Júlia for their support, their love, and their patience throughout this continuing common experience. Their emotional support and understanding have been and will always be the striking force in my life.

This research was supported in part by the Portuguese INVOTAN Committee.

Contents

1 Introduction **15**

2 Generic Circuit Simulation **19**

2.1 Introduction 19

2.2 System Equation Formulation and Solution 19

2.3 Direct Method Circuit Simulation 20

2.4 Relaxation-Based Circuit Simulation 21

2.5 Parallelism Issues 27

3 Increasing Parallelism in Circuit Simulation **31**

3.1 Introduction 31

3.2 Explicit Numerical Integration 31

3.2.1 Explicit Exponential Fitting Integration Formulas 32

3.2.2 Large Timestep Properties of Exponential Fitting Formulas 35

3.2.3 Effects of Equation Ordering 44

3.2.4 Discussion of Advantages and Drawbacks 47

3.3 Multiple Timepoint Algorithms 49

3.3.1 The Waveform Newton Algorithm 50

3.3.2 Waveform Newton Circuit Simulation 53

3.3.3 Implementation Issues and Discussion of Results 56

4 The Connection Machine **63**

4.1 Architectural Overview 63

4.2 Virtual Processors 66

4.3 Data Parallelism 66

4.4 Programming the Connection Machine 68

5 Massively Parallel Circuit Simulation **73**

5.1 Introduction 73

5.2 Generic Circuit Simulation on the Connection Machine 73

5.2.1 Nonlinear Band Relaxation 75
5.2.2 A Parallel Band-matrix Solver 78
5.2.3 Non-Linear Band-Relaxation on the Connection Machine 85
5.2.4 Discussion of Results 90
5.3 Simulating Analog Grids on the Connection Machine 94
5.3.1 Circuits for Early Vision Tasks 95
5.3.2 Parallel Simulation of Vision Circuits on the Connection Machine 102
5.3.3 Simulation Results 108

6 Conclusions **113**
6.1 Summary of Results 113
6.2 Future Work 114

A The Simlab Simulation Environment **117**

List of Figures

2-1 Parallel Model Evaluation 28

3-1 Updating a node value with CINNAMON (circuit interpretation) 34
3-2 Test Circuit for Explicit Integration Formulas 35
3-3 Voltage in node 1 for $h = 0.1$ 35
3-4 Voltage in node 1 for $h = 1.1$ 36
3-5 Voltage in node 1 for $h = 10$ 37
3-6 Circuit used for determining $\mathbf{x}_\infty$ for CINNAMON, XPSim and SI 45
3-7 Voltage in node 1 for $h = 1.1$ 46
3-8 Circuit to test possible stability of the ordered XPSim algorithm 46
3-9 $v_1(t)$ from the circuit in Figure 3-8 for $h = 6$ 47
3-10 $v_1(t)$ from the circuit in Figure 3-2 for $h = 2, (v_1(0) = 0.8V)$ 48

4-1 CM data processor. (Copyright *Thinking Machines Corporation*. Used with permission) 64
4-2 The Connection Machine Computational Model. 65
4-3 Virtual processing on the *Connection Machine*. (Copyright *Thinking Machines Corporation*. Used with permission) 66
4-4 Data-parallelism on the *Connection Machine*. (Copyright *Thinking Machines Corporation*¡. Used with permission) 68

5-1 Illustration of a step of the Cyclic Reduction algorithm. 78
5-2 Block Tridiagonal Matrix showing partitioned blocks. 79
5-3 Example block tridiagonal matrix with 4 partitions of 4 row-blocks each. 80
5-4 Example matrix after step 1 of Wang's algorithm. 80
5-5 Example matrix after step 2 of Wang's algorithm. 81
5-6 Example matrix after the first part of step 3 of Wang's algorithm. 81
5-7 Example matrix after step 3 of Wang's algorithm. 82
5-8 Example matrix after the first part of step 4 of Wang's algorithm. 82
5-9 Example matrix after step 4 of Wang's algorithm. 82

5-10 Data-Structures used in the Band-Relaxation Circuit Simulator implementation . 87
5-11 Grid of Linear Resistors. 96
5-12 Grid of Linear Resistors with capacitor to ground at each node. 97
5-13 Grid of nonlinear resistors with capacitor to ground. 98
5-14 Nonlinear resistor i-v Characteristic. 99
5-15 Division of the circuit into identical cells. 104
5-16 Separation of the cells by duplicating nodes (capacitors to ground omitted for simplicity). 105
5-17 Circuit interpretation for maintaining node voltage consistency (capacitors to ground omitted for simplicity). 106
5-18 Circuit interpretation for maintaining node current consistency (capacitors to ground omitted for simplicity). 106

List of Tables

3-1 Circuit Data for the test circuits. 57
3-2 Simulation data with the 2^{nd} order BDF. 58
3-3 Simulation data with waveform-newton using the 2^{nd} order BDF. 58

4-1 Using Matrix-Vector products to measure primitive operations. 70
4-2 Measuring speedup, raw performance and scalability with Virtual Processor ratio. 71

5-1 Sequential Complexity of the Modified Wang algorithm. 83
5-2 Parallel Complexity of the Modified Wang algorithm. 84
5-3 Performance Results for the Band-Solver Algorithm. 90
5-4 Number of reads from other processor's memory. 92
5-5 Non-Linear Band-Relaxation Circuit Simulation results. 92
5-6 Linear Resistor Grid Experiments. 108
5-7 DC solution results for the linear resistor grid experiments. 109
5-8 Nonlinear resistor grid experiments. 109
5-9 Results of the nonlinear resistor grid experiments. 110

1

Introduction

The ever increasing size and complexity of today's integrated circuit designs and the continuous development of processing technologies has led to an increase in the fabrication costs and development time of such designs. In order to minimize such growing costs, accurate and reliable verification of each project before fabrication becomes imperative, so that fabricated circuits function correctly on "first silicon".

As circuits become larger and more complex, design techniques become more elaborate and innovative in order to fully exploit the available space. Therefore, it becomes necessary that accurate circuit simulation methods be applied instead of other more intuitive but less detailed approaches to verification, such as timing simulation or functional simulation. However, full verification of the functionality of a system must not only be reliable, but also fast enough to provide results in an acceptable time interval. This implies that the computational resources that must be allocated for simulation also increase dramatically.

Transient circuit simulation involves forming the system of nonlinear differential equations that describes the dynamics of a circuit and solving them numerically. The integration methods used to solve the circuit equations require several evaluations of all the devices in the circuit and several sparse matrix decompositions and solutions at each discrete timepoint. Conventional circuit simulators, developed during the 1970's, like SPICE [Nagel75] or ASTAP [Weeks73] were aimed at simulating circuits with a few hundred devices. The more recent need to simulate larger and more complex circuits has led to computational problems, mostly because of the large number of complex device evaluations required and the fact that the complexity of standard direct matrix solution methods grows superlinearly with the size of the problem. More recent algorithmic developments, such as the application of relaxation methods, can in certain cases reduce the complexity to $O(N)$ (where N is the size of the problem) but this may still be too large for big problems [White86].

Improvements in computer architectures will allow faster simulation to be performed with existing algorithms, but while the size and complexity of circuits continues to increase, existing

architectures are approaching their technological and physical limits. The expected computational needs to simulate circuits with, say, 10^6 devices is clearly outside of the possibilities of current machine architectures.

Several techniques for algorithm speedup have been pursued: these include generation of special-purpose microcode for equation solution, table-lookup model evaluation, use of vector processor machines, etc. It becomes clear, however, that such approaches will not provide sufficient increase in the computer performance and that parallel computer architectures must be devised and algorithms for their efficient usage must be developed to keep pace with the needed performance. The more recent development of general purpose parallel processing computers has made parallelism widely accepted as the next step to be adopted for hardware and algorithm acceleration [Hancock86]. Such machines require the development of new algorithms specifically targeted to their architectures to be able to efficiently exploit the resources made available.

Numerous implementations of circuit simulators on parallel processing machines have been attempted [Deutsch85, Gallivan, Smart87, White85]. However, the parallel computers used were devised as mere extensions of the Von Neumann paradigm, consisting of the addition of a few more processors to work on the same problem simultaneously. Therefore early parallel circuit simulators have been developed on coarse-grained parallel architectures or machines with a small number of processors (on the order of 10). The recent availability of massively parallel computers, machines with a number of processors on the order of thousands, has broadened the range of possibilities in the field of parallel circuit simulation. Such machines are an even greater departure from conventional uniprocessor architectures, and specific algorithms targeted to their special architectures must be developed if one is to take advantage of the massive number of computational units available. The *Connection Machine*, a 65K processor supercomputer built by Thinking Machines Corporation [Hillis85], is an example of such an architecture. This machine's characteristics make it a very promising machine for computationally intensive problems such as circuit simulation, provided the available computational power can be used effectively.

To investigate the problem of an efficiently implementation of a circuit simulator on a massively parallel computer, we start by determining which features of the problem make it desirable for such a computing model. Studying possible solutions to those simpler problems leads to a deeper understanding of the main issues involved in dealing with the kind of architectural constraints imposed by the machine.

The *Connection Machine* is a massively parallel supercomputer based on the Data Parallelism paradigm [CMDataPar]. The machine's control structure is Single Instruction on Multiple Data (SIMD) [Flynn66, Hwang84] and therefore all processors at any time execute the same instruction, although some may be optionally idle. To first approximation, this structure seems interesting for the type of problems in which the amount of data to be worked upon is

large compared to the necessary computations to be performed on it, or equivalently, for those problems that exhibit inherent parallelism at the data level.

Circuit simulation is a very computational expensive problem presenting some features that make it attractive for an implementation on a massively parallel processor. The amount of data to be dealt with can be very large, and the data can be divided onto a small number of classes each of which requires the same computation. The two main computational intensive tasks in a circuit simulator are the evaluation of the nonlinear circuit elements and the solution of a sparse matrix system resulting from discretization and linearization of the system of differential equations. For circuit simulation to be efficient on this kind of architecture, both problems must be dealt with effectively.

It is agreed upon by most computer architects that communication cost is a first order factor to be considered in any program implementation in a parallel computer. Such a fact has been verified on the *Connection Machine*. Recent performance benchmarks [Levit88] have indicated that the cost of communication can overtake by a factor of $2 \rightarrow 30$ the cost of executing a floating-point operation. Moreover, the factor of 2 is related to short distance communication using the two dimensional grid primitives available on the *Connection Machine*, while the factor of 30 refers to generic random communication over long distances using a router network. This seems that the *Connection Machine* will be best for simulating circuits which exhibit high regularity and locality.

In this thesis, several new solution methods are investigated and the results confirm that some of the constraints imposed by the *Connection Machine* architecture, namely the cost of communication and the SIMD nature of the machine, make general circuit simulation inefficient.

On the other hand, good efficiency can be achieved by restricting the class of circuits to those which exhibit inherent structural regularity can help to improve circuit simulation performance. In particular, circuits for early vision tasks, such as smoothing and segmentation, form a class of circuits which, for the most part, cannot be simulated with traditional programs on traditional computers. These circuits are necessarily very large and their simulation must be accomplished at the analog level due to the high accuracy required. Typical electrical circuit simulators are not able to handle vision circuits simply because of their immense size, since the computation time for these simulators grows super-linearly with the size of the circuit.

This thesis consists of five chapters and one appendix. After this introductory chapter, Chapter 2 reviews the problem of generic circuit simulation, introduces the algorithms used and motivates studying new algorithms for parallel implementation of circuit simulators. Next, in Chapter 3, we study two approaches for exploiting parallelism in circuit simulation. We look at explicit integration algorithms, with emphasis on exponential fitting techniques. An approach for increasing the parallelism exploitable in the problem is also investigated which is based upon studying solutions in a waveform space, therefore using multiple timepoint algorithms. Chapter 4 follows with a brief description of the target architecture of the implementations

developed. The main architectural and environmental features of the *Connection Machine* are described and its programming model is presented. Next, in Chapter 5, our efforts in implementing generic circuit simulation tools on the *Connection Machine* are described. We discuss a band-relaxation approach pursued in the hope of improving the convergence properties of relaxation methods while still maintaining high inherent parallelism. Our understanding of the constraints imposed by the architecture of the machine provides the framework for the study of techniques aimed at efficiently simulating vision circuits. Our study focus on a representative set of vision circuits, describing the tasks they are associated with, and discusses their unique features. Finally we present strategy developed for the implementation of an efficient circuit simulation program on the *Connection Machine* that can obtain in excess of 1 Gigaflop performance. The end of Chapter 5 is devoted to numerical results for several vision circuits simulated with different input images. The conclusion and suggestions for further work follow in Chapter 6. Finally appendix A briefly describes the SIMLAB circuit simulation environment as it was the framework for most of the research described in this thesis.

2

Generic Circuit Simulation

2.1 Introduction

Electrical circuit simulation begins with the formulation of the system of ordinary nonlinear differential equations which govern its behavior. These equations are introduced in section 2.2 and the common algorithms used to compute their solution are also described. Specifically, direct methods based on some form of sparse Gaussian-elimination for the solution of the linear systems obtained at each timepoint, and Relaxation methods, are briefly reviewed in sections 2.3 and 2.4. In keeping with our goal of exploiting parallelism in the circuit simulation problem, in section 2.5 we look into the various steps of the simulation process and try to determine the problems they present to a parallel implementation. This involves comparing the relative strengths of explicit and implicit integration methods and the possible advantages of relaxation algorithms.

2.2 System Equation Formulation and Solution

The electrical behavior of a circuit in the time domain can be described by a system of equations involving node voltages, resistive currents, capacitor charges and inductor fluxes. This system of equations can be constructed from a circuit description using the Kirchoff current and voltage Laws (KCL and KVL) and the constitutive or branch equations of each circuit element. If we restrict ourselves to noninductive circuits, we can use a technique referred to as nodal analysis [Desoer69, Chua87, Vlach83] to formulate those equations.

Assuming for simplicity that all resistive elements, including active devices, are characterized by constitutive equations where voltages are the controlling variables and currents are the controlled variables, that energy storage elements are represented by voltage-controlled charge devices and that all voltage sources have one terminal connected to ground or can be transformed into current sources with the use of the Norton transformation, then nodal analysis

yields a system of equations of the form:

$$\frac{d}{dt}\mathbf{q}\left(\mathbf{v}(t),\mathbf{u}(t)\right)+\mathbf{f}\left(\mathbf{v}(t),\mathbf{u}(t)\right)=\mathbf{0} \qquad \mathbf{v}(0)=\mathbf{v}_0. \tag{2.1}$$

Here n is the number of circuit nodes excluding the reference, $\mathbf{q}\left(\mathbf{v}(t),\mathbf{u}(t)\right)\in\Re^n$ is the vector of sums of capacitive charges at each node, $\mathbf{f}\left(\mathbf{v}(t),\mathbf{u}(t)\right)\in\Re^n$ is the vector of sums of resistive currents at each node, $\mathbf{u}(t)\in\Re^n$ is the vector of input voltages, and $\mathbf{v}(t)\in\Re^n$ is the vector of node voltages with initial values $\mathbf{v}(0)=\mathbf{v}_0$. Each equation in this system corresponds to applying KCL to each node in the circuit and using the constitutive equations of the branch elements to express their currents in terms of the node voltages.

It is possible to extend the nodal analysis technique to include circuits with inductors and voltage sources by using modified nodal analysis [Ho75], while still preserving the form of Eqn. (2.1) by changing the physical meaning of some of the unknowns, but we will not follow that path. However, many of the comments made in the following about these systems apply to any system in the form of Eqn. (2.1).

2.3 Direct Method Circuit Simulation

To solve the systems of nonlinear ordinary differential equations (ODE's) in Eqn. (2.1) in the interval $t\in[0,T]$, the interval is discretized and a numerical solution is computed at each point of the discretization. This is usually accomplished using Linear Multistep Formulas [Dahlquist74, Trefethen89] which when applied to the first order ordinary differential equation

$$\frac{d}{dt}x(t)=f(x(t),t) \qquad x(0)=x_0 \qquad t\in[0,T], \tag{2.2}$$

have the general form

$$\sum_{i=0}^{s}\alpha_i y(t_{m+1-i})=h\ \sum_{i=0}^{s}\beta_i f(y(t_{m+1-i}),t_{m+1-s}) \tag{2.3}$$

where h is the discretization timestep, $y(t_m)$ is the estimated value of the solution at time $t=t_m=hm$, $f(y(t_{m+1-i}),t_{m+1-s})$ represents the dynamics of the equation at the given timepoint with the estimated value and the parameters α_i e β_i are chosen for accuracy within stability limits and s is the formula's order. If $\beta_0\neq 0$ in Eqn. (2.3), the method is an *implicit method*, otherwise it is an *explicit method*. Implicit methods are far more stable than their explicit counterparts, therefore allowing the use of larger timesteps for stiff problems, but unfortunately give rise to systems of coupled equations, which are expensive to solve.

In circuit simulation, the trapezoidal formula is frequently used to approximate the system of Eqn. (2.1) by a sequence of implicit algebraic equations. Given a timestep, h, the trapezoidal integration algorithm is defined as:

$$y(t_{m+1})=y(t_m)+\frac{h}{2}\left(f(y(t_{m+1}),t_{m+1})+f(y(t_m),t_m)\right) \tag{2.4}$$

When applied to Eqn. (2.1) the trapezoidal formula yields:

$$\mathbf{q}\left(\mathbf{v}(t+h),\mathbf{u}(t+h)\right)-\mathbf{q}\left(\mathbf{v}(t),\mathbf{u}(t)\right)+\frac{h}{2}\left[\mathbf{f}\left(\mathbf{v}(t+h),\mathbf{u}(t+h)\right)+\mathbf{f}\left(\mathbf{v}(t),\mathbf{u}(t)\right)\right]=\mathbf{0} \tag{2.5}$$

where $\mathbf{v}(t)$ is known, and the equation must be solved to compute $\mathbf{v}(t+h)$.

If an iterative method is used to solve Eqn. (2.5) for $\mathbf{v}(t+h)$, the residue at the k^{th} step, $\mathbf{F}\left(\mathbf{v}^k(t+h)\right)$, is defined as

$$\begin{aligned}\mathbf{F}\left(\mathbf{v}^k(t+h)\right)=\mathbf{q}(\mathbf{v}^k(t+h),\mathbf{u}(t+h))-\mathbf{q}\left(\mathbf{v}(t),\mathbf{u}(t)\right)-\\ \tfrac{h}{2}\left(\mathbf{f}\left(\mathbf{v}^k(t+h),\mathbf{u}(t+h)\right)+\mathbf{f}\left(\mathbf{v}(t),\mathbf{u}(t)\right)\right)\end{aligned} \tag{2.6}$$

and the Jacobian of $\mathbf{F}\left(\mathbf{v}^k(t+h)\right)$, $\mathbf{J}_F\left(\mathbf{v}^k(t+h)\right)$ is then given as

$$\mathbf{J}_F\left(\mathbf{v}^k(t+h)\right)=\frac{\partial\mathbf{q}\left(\mathbf{v}^k(t+h),\mathbf{u}(t+h)\right)}{\partial\mathbf{v}}-\frac{h}{2}\frac{\partial\mathbf{f}\left(\mathbf{v}^k(t+h),\mathbf{u}(t+h)\right)}{\partial\mathbf{v}}. \tag{2.7}$$

Then, if Newton's method is used to solve Eqn. (2.5), $\mathbf{v}^{k+1}(t+h)$ is derived from $\mathbf{v}^k(t+h)$ by solving, the linear system of equations

$$\mathbf{J}_F\left(\mathbf{v}^k(t+h)\right)\left[\mathbf{v}^{k+1}(t+h)-\mathbf{v}^k(t+h)\right]=-\mathbf{F}(\mathbf{v}^k(t+h)). \tag{2.8}$$

The Newton iteration is continued until $\|\mathbf{v}^{k+1}(t+h)-\mathbf{v}^k(t+h)\|<\epsilon$, where $\epsilon>0$ is a small number and $\mathbf{F}(\mathbf{v}^k(t+h))$ is close enough to zero. Eqn. (2.8) is now a system of linear equations which can be solved using some form of sparse Gaussian elimination (GE).

The above algorithm is known as the *Direct Method* for solving Eqn. (2.1) and is summarized in Algorithm 2.1.

As is clear from Eqn. (2.8), using the direct method to simulate large circuits implies solving large linear systems, usually with some form Gaussian elimination. In general, dense matrix solution by Gaussian elimination grows in complexity *like* n^3 where n is the number of unknowns in the system. This rapid growth in complexity has focused attention on finding other approaches for solving large linear systems. Since the matrices that describe the linear systems in circuit simulation problems are usually sparse, that is, most of matrix entries are zero, clever data structures and careful reordering algorithms have been used to reduce the complexity of Gaussian elimination for sparse matrices to less than n^2 [Kundert86]. However, sparse matrix techniques based on Gaussian elimination still grow superlinearly with the number of unknowns, and therefore researchers have continued to seek matrix solution methods that do not grow as rapidly.

2.4 Relaxation-Based Circuit Simulation

Relaxation methods comprise one class of iterative solution methods for systems of equations which grow more slowly with problem size than Newton's method followed by sparse Gaussian

Algorithm 2.1 (Direct Method Circuit Simulation)

```
simulate()
{
  t = 0 ;
  formulate the circuit equations ;
  while (t < Simulation_Time) {
    choose h ;
    repeat {
      use Newton's method to solve the nonlinear algebraic equations;
      solve the resulting linear algebraic system with sparse GE;
    }
    until (converged) ;
    t = t + h ;
  }
}
```

elimination. In particular, the computation per iteration grows linearly with the number of nonzero jacobian matrix entries. The disadvantage of relaxation is that each iteration only produces an *approximate* solution to a linear system, and repeated relaxation iterations do not necessarily improve that approximation. For some circuit simulation problems, relaxation methods work well because repeated relaxation iterations converge rapidly to the exact solution [Newton83].

Relaxation methods can be used to solve Eqn. (2.1) in a number of ways. Applying relaxation to the solution of Eqn. (2.1) has in general a simple circuit interpretation. For instance, when updating the voltage at a particular node, one could consider all the other nodes in the circuit as constant voltages sources. One could then solve the corresponding scalar equation for that node voltage and afterwards use the same procedure for the other nodes. If this is performed in an iterative fashion, then one could hope that after a certain number of iterations a convergent set of node voltages would be obtained. Viewed in this way, relaxation method perform some decoupling of the equations or correspondingly performs some partitioning of the circuit. The two most common methods of relaxation used in electrical circuit simulation are the Gauss-Jacobi method and the Gauss-Seidel Method [Ortega70] and they follow exactly the procedure described. They differ only in the sense that at each iteration the Gauss-Jacobi method only uses values from previous iterations, while the Gauss-Seidel method uses the most recent updates for each node's voltage, which is the reason why Gauss-Seidel has a faster convergence than Gauss-Jacobi for most problems.

Relaxation methods can be applied at different stages of the solution process. If applied to the linear equation (2.8) these methods are known as Linear Relaxation (LR) [Varga62]. If applied to the nonlinear equation (2.5) they are called Nonlinear Relaxation (NLR) [Ortega70] and if applied to the differential equation (2.1) directly they are known as Waveform Relaxation (WR) [White86].

Under very mild conditions, the three relaxations techniques mentioned can be proven to be convergent. We will not dwell deeply into the properties of these methods. However for reasons of completeness, we include theorems and more detailed descriptions for the Gauss-Jacobi versions of the relaxation methods considered, as we will refer to them later in this thesis.

Linear Relaxation can be used as a substitute for sparse Gaussian-elimination in the solution of the linear problem of Eqn. (2.8). More generally, when applied to a linear system of the form $\mathbf{A}\,\mathbf{x} - \mathbf{b} = 0$ where $\mathbf{x} = (x_1, ..., x_n)^T$, $\mathbf{b} = (b_1, ..., b_n)^T$, $x_i, b_i \in \Re$, and $\mathbf{A} = (a_{ij})$, $A \in \Re^{n \times n}$, the Gauss-Jacobi relaxation method attempts to solve it by solving the equations one row at time, using for the other variables the values from previous iterations. This is shown in Algorithm 2.2, where the superscript k is the iteration count and ϵ is a small positive number.

Algorithm 2.2 (Linear Gauss-Jacobi Relaxation for solving $\mathbf{A}\,\mathbf{x} - \mathbf{b} = 0$)

linear_Gauss_Jacobi_relaxation()
{
  $k = 0$;
  guess some $\mathbf{x}^0$
  repeat {
    $k = k + 1$;
    forall $(i \in \{1, .., n\})$ {
      $x_i^k = \frac{1}{a_{ii}} \left[b_i - \sum_{j=1}^{i-1} a_{ij} x_i^k + \sum_{j=i+1}^{n} a_{ij} x_i^{k-1} \right]$;
    }
    until $(\|\mathbf{x}^k - \mathbf{x}^{k-1}\| \leq \epsilon)$
  }
}

The following theorem illustrates the necessary conditions for convergence regardless of the initial guess. A proof can be found in [Golub83, White86].

LINEAR GAUSS-JACOBI RELAXATION CIRCUIT SIMULATION

Theorem 2.1 *Let the Gauss-Jacobi relaxation algorithm be used to solve for the linear system of equations in Eqn. (2.8) generated at each Newton iteration. If* $\mathbf{f}(\mathbf{v}, \mathbf{u})$ *is continuously differentiable with respect to* $\mathbf{v}$, $\mathbf{u}(t)$ *is continuous with respect to* t, $\frac{\partial \mathbf{q}}{\partial \mathbf{v}}(\mathbf{v}, \mathbf{u})$ *is strictly diagonally dominant then there exists an* $\mathbf{h}$ *such that for all* $h_m \leq \mathbf{h}$ *the relaxation converges to the solution of Eqn. (2.5).*

In terms of circuit topology, the above sufficient conditions are met if there is a capacitor to ground at each node, an assumption that is reasonable in VLSI designs. For the case of Nonlinear Relaxation the following theorem illustrates the necessary conditions for convergence regardless of the initial guess. A proof can be found in [Ortega70, White86] and shown to be independent of the linear multistep formula used.

NONLINEAR GAUSS-JACOBI RELAXATION CIRCUIT SIMULATION

Theorem 2.2 *Let the Gauss-Jacobi Nonlinear Relaxation algorithm be used to solve for* $\mathbf{v}(t+h)$ *in Eqn. (2.5). If* $\mathbf{f}(\mathbf{v}, \mathbf{u})$ *is continuously differentiable with respect to* $\mathbf{v}$ *uniformly in* $\mathbf{u}$, $\mathbf{u}(t)$ *is continuous with respect to* t, $\frac{\partial \mathbf{q}}{\partial \mathbf{v}}(\mathbf{v}, \mathbf{u})$ *is strictly or irreducibly diagonally dominant uniformly over all* $\mathbf{v}$ *and* $\mathbf{u}$, *and* $\mathbf{v}(t)+O(h)$ *(where* $O(h)$ *is continuous in* h *and* $\lim_{h\to 0} O(h) = 0$*) is used as an initial guess for the relaxation, then there exists an* $\mathbf{h}$ *such that for all* $h_m \leq \mathbf{h}$ *the relaxation converges to the solution of Eqn. (2.5).*

The point Gauss-Jacobi Nonlinear Relaxation method applied at the nonlinear equation (2.5) is shown in Algorithm 2.3. The superscript k represents the iteration counter, the subscript i represents the node index in the vector of voltages, and ϵ is a small positive number. The function $F_i(\cdot)$ is the i^{th} component of equation (2.5), and v_i^k is the i^{th} component of $\mathbf{v}(t+h)$ at the k^{th} relaxation iteration.

It is possible to use the Newton-Raphson algorithm to solve the algebraic equation

$$F_i\left(v_1^{k-1}, \ldots, v_i^k, \ldots, v_N^{k-1}\right) = 0$$

accurately at each step, but this is not necessary. In fact it has been proved that only one iteration of the Newton-Raphson method is required to maintain the convergence of the relaxation iteration due to the quadratic convergence of the Newton-Raphson method [Newton83]. After the relaxation iteration has converged, $\mathbf{v}^k$ becomes the new voltage vector at time $t+h$, and one advances to the next timepoint.

One of the reasons why the relaxation-Newton methods are popular for circuit simulation is that for a broad class of circuits the capacitance matrix defined as

$$\mathbf{C}\left(\mathbf{v}(t), \mathbf{u}(t)\right) = \frac{\partial}{\partial \mathbf{v}} \mathbf{q}\left(\mathbf{v}(t), \mathbf{u}(t)\right) \tag{2.9}$$

Algorithm 2.3 (Nonlinear Gauss-Jacobi Relaxation)

nonlinear_Gauss_Jacobi_relaxation()
{
  $t = 0$;
  while *(t < Simulation_Time)* {
    chooseh ;
    $k = 0$;
    guess $\mathbf{v}^0$
    repeat {
      $k = k + 1$;
      forall $(i \in \{1, .., n\})$ {
        solve $F_i\left(v_1^{k-1}, \ldots, v_i^k, \ldots, v_N^{k-1}\right) = 0$ *for* v_i^k, *given* v_j^{k-1} $j \neq i$
      }
    }
    until $\left(\max_{1 \leq i \leq N} \left|v_i^k - v_i^{k-1}\right| \leq \epsilon\right)$
    update waveforms ;
    $t = t + h$;
  }
}

is strictly diagonally dominant. This is the case if the circuit contains two-terminal positive capacitors (or any other elements whose charge function has a diagonally dominant Jacobian with positive diagonal entries), and there exists some nonzero capacitance or a voltage source to ground at each node in the circuit. Therefore, for these circuits, the Relaxation-Newton algorithms are guaranteed to converge if the integration timestep is made small enough.

Applying relaxation directly to Eqn. (2.1) leads to Waveform Relaxation. To derive the algorithm, we simply apply the desired relaxation iteration principle directly to the system of differential equations. We illustrate the approach by applying the Gauss-Jacobi method to Eqn. (2.1) as shown in Algorithm 2.4 where the superscript k represents the iteration counter, the subscript i represents the equation index, ϵ is a small positive number and $\tilde{\mathbf{v}}_i^{k+1}(\cdot) \equiv (v_1^k, \cdots, v_{i-1}^k, v_i^{k+1}, v_{i+1}^k, \cdots v_n^{k-1})^T$.

Algorithm 2.4 (Gauss-Jacobi Waveform Relaxation)

Gauss_Jacobi_waveform_relaxation()
{
  $t = 0$;
  $k = 0$;
  forall $(i \in \{1, .., n\})$ {
    guess $v_i^0(t)$ *for all* $t \in [0, T]$
  }
  repeat {
    $k = k + 1$;
    forall $(i \in \{1, .., n\})$ {
      solve $\frac{d}{dt} q_i \left(\tilde{\mathbf{v}}_i^k(t), u(t) \right) - f_i \left(\tilde{\mathbf{v}}_i^k(t), u(t) \right) = 0$
    }
  until $\left(\max_{i \in \{1,..,n\}} \left| v_i^k - v_i^{k-1} \right| \leq \epsilon \right)$
  }
}

In this approach, at each step of the relaxation, the *waveform* $v_i(t)$ for all $t \in [0, T]$ is updated by solving the associated differential iteration equation independently from the other equations. This allows the use of an independent set of integration timesteps. Therefore, the Waveform Relaxation algorithm is inherently a multirate integration method.

The convergence of WR is guaranteed for most examples of practical interest by theorem 2.3 which however only applies to the continuous case. For proofs, results on the discretized case and additional insight see [Lelarasmee82, White86].

GAUSS-JACOBI WAVEFORM RELAXATION CIRCUIT SIMULATION

Theorem 2.3 *If, in a system of the form of Eqn. (2.1),* $\mathbf{f}(\mathbf{v}, \mathbf{u})$ *is Lipschitz continuous with respect to* $\mathbf{v}$ *uniformly in* $\mathbf{u}$, $\mathbf{u}(t)$ *is continuous with respect to* t *and* $\frac{\partial \mathbf{q}(\mathbf{v},\mathbf{u})}{\partial \mathbf{v}}$ *is strictly or irreducibly diagonally dominant uniformly over all* $\mathbf{v}$ *and* $\mathbf{u}$, *then the Gauss-Jacobi Waveform Relaxation algorithm applied to such a system will converge to the solution of the system*

2.5 Parallelism Issues

Besides improved algorithmic efficiency, one way of improving simulator speed is to perform some computations simultaneously. This can be achieved with parallel processing. Let us consider the computational intensive tasks that must be performed in a simulation and determine how they can be parallelized.

If one is using direct methods, the two computationally intensive phases of the simulation process are the evaluation of the model elements and the solution of the linear system of equations. In general, the model evaluation will grow linearly with the number of elements in the circuit, whereas the linear system solution will grow super-linearly with the number of nodes in the circuit. Typically, in a circuit with more than about 3,000 nodes, the linear system solution will dominate computation time.

Consider first the device evaluations; these are to some extent always necessary, regardless of the simulation algorithm chosen. If we assign each device (or groups of devices) in the circuit to a separate processor, the device evaluations can be done in parallel. terms in the jacobian and Newton residue must also be updated. If the storage for these terms reside somewhere in common memory, they can be updated in parallel, provided there is some mechanism for handling contentions. As illustrated in Figure 2-1, the MOS transistors can be evaluated in parallel, their currents calculated and the entries corresponding to their terminal nodes, subsequently stored and merged if necessary. The two leftmost transistors for instance, can be evaluated in parallel and the Newton residue for their common node is calculated as the sum of the currents flowing from the devices into the node.

Consider the linear system solution which is necessary in direct methods circuit simulation. Gaussian-elimination does not have a large amount of inherent parallelism and are not readily parallelized when using fine- or medium-grain architectures [Jacob86]. However, there are effective methods for parallelizing relaxation-based linear system solvers (see [Lumsdaine88, White86]). Furthermore, these same methods can be applied to the nonlinear relaxation-based simulators.

In considering relaxation methods, the main advantage they present is that they perform some decoupling of the equations as we saw earlier. This decoupling makes it easier to exploit parallelism by having different equations solved in different computational units. This is shown for instance in algorithm 2.2 where the **forall** statement implies that each of the node

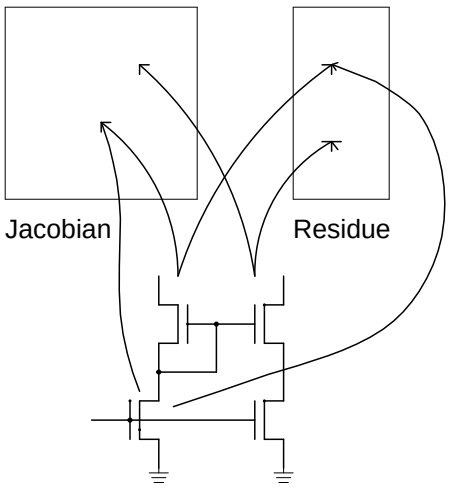

FIGURE 2-1: Parallel Model Evaluation

voltages can be updated independently, i.e., in parallel. Moreover, if the decoupling of the equations is seen as a form of dynamic circuit partitioning, then relaxation methods further extend the parallelism exhibited by the problem by solving smaller tightly coupled subproblems simultaneously.

Direct methods, as we noted above, present the problem that some form of Gaussian elimination must be used to solve the large sparse system of linear equations, and there are several difficulties associated with parallelizing sparse techniques. However, this problem could possibly be avoided in its entirety, if it were not necessary to construct such a matrix. This possibility can in fact be made real if the discretization of the circuit differential equations, Eqn. (2.1), is done using an *explicit* Linear Multistep Formula.

To illustrate the possible advantage of using explicit linear multistep formulas when discretizing ordinary differential equations, we recall the forward-Euler integration formula, which given a timestep h, is defined as

$$y(t_{m+1}) = y(t_m) + h\ f(y(t_m), t_m). \tag{2.10}$$

For comparison purposes we also recall the simplest implicit formula, the backward-Euler, also known as implicit-Euler formula which given a timestep h, is defined as

$$y(t_{m+1}) = y(t_m) + h\ f(y(t_{m+1}), t_{m+1}) \tag{2.11}$$

Comparison of Eqns. (2.10) and (2.11) reveals the advantages of explicit multistep formulas. In order to calculate the values of the unknowns at the new time point, no system solution is necessary, function evaluations suffice. Clearly all this added simplicity is not without cost: for one, implicit methods are typically more stable than explicit methods and this usually allows an implicit method to take larger timesteps. In some cases, explicit methods are still attractive since a function evaluation is usually computationally less expensive than a system solution.

In these cases, the system must be such that stability is not a problem. However, we note that Eqn. (2.10) implies that the values of all the unknowns can be calculated simultaneously, i.e. in parallel. The possible advantages are therefore so overwhelming that explicit integration formulas are a topic of some research in the context of parallel circuit simulation.

3

Increasing Parallelism in Circuit Simulation

3.1 Introduction

In this chapter we present our study of some techniques and algorithms aimed at more efficiently exploiting the parallelism inherent in the circuit simulation problem. Explicit numerical integration techniques are investigated in section 3.2 in order to determine their suitability for the simulation problem in the context of massively parallel processing. Section 3.3 presents an approach based on exploiting multiple timepoint algorithms in order to increase the parallelism in the simulation problem, from which some preliminary conclusions are attempted.

3.2 Explicit Numerical Integration

The possible advantage of using explicit integration methods was motivated in Section 2.5 with the well-known forward-Euler formula (see Eqn. (2.10). The properties of explicit linear multistep formulas have been known for a long time now [Dahlquist74]. Their stability is in general reduced and decreases with increasing order. For these reasons these integration methods do not seem very adequate for circuit simulation problems, since stability and accuracy requirements together will imply the use of exceedingly small timesteps.

Designers of MOS digital circuits often use transistor-level simulation programs that are very fast but have limited accuracy when compared to circuit simulation programs like SPICE [Nagel75]. Programs of this type are referred to as *timing simulators*, and typically are simplified circuit simulators with loosely controlled accuracy. Specifically, these programs use nodal analysis to derive a system of differential equations that describes the circuit, and then by exploiting the assumption that each node has a capacitor to ground, can use simplified multistep integration algorithms [Chawla75, Saleh83]. The assumption that the only capacitors in the

circuits are capacitors to ground at each node, is sufficiently accurate in most cases. MOS circuit however, present a high-degree of stiffness that is usually present in the diagonal terms of the jacobian, and that requires for a method to at least be implicit with respect to the diagonal terms. This excludes the forward-Euler method.

In this study we will restrict ourselves to a class of first order explicit integration formulas, namely explicit exponential fitting integration formulas [Neto90]. These formulas may potentially perform better than most of the integration algorithms used in conventional timing simulators. Furthermore, they present lower complexity and from our point of view retain the attractive prospectives that they present high-degree of parallelism.

3.2.1 Explicit Exponential Fitting Integration Formulas

Recently, exponentially-fit integration methods [Sarkani74] have been reinvestigated in an attempt to improve the performance and accuracy of timing simulation, as in the programs CINNAMON [Vidigal86], XPSim [Bauer88] and Elogic [Kim88]. For the purposes of this study, we define the first-order exponentially-fit integration algorithm as

$$x(t_{m+1}) = x(t_m) + (x(\infty) - x(t_m))\,(1 - e^{-\frac{h}{\tau}}) \tag{3.1}$$

where $h = t_{m+1}t_m$ and the $x(\infty)$ and τ depend on the precise exponentially-fit integration method being used. The interpretation of these parameters is that $x(\infty)$ is an estimate of the equilibrium or steady-state value of x, and τ is an estimate of the time-constant of the approach to steady-state.

Exponentially-fit methods are appealing in that when applied to numerically integrating the scalar linear differential equation $\frac{d}{dt} + dx = b$, $d, b \in \Re$, with an appropriate choice of τ and $x(\infty)$, the exact solution is produced, **no matter how large the timestep**. It is conjectured that this accuracy for scalar problems has the practical consequence that exponentially-fit integration methods retain reasonable accuracy on general problems for much larger timesteps than standard multistep methods. The theoretical justification for this large timestep behavior was however until now very limited.

Not all values of the $x(\infty)$ and τ parameters introduced in Eqn. 3.1 produce consistent integration methods, where by consistency we mean that the error introduced in one timestep, h, is $O(h^2)$. Assuming that $x(t_m)$ is exact and using a Taylor series expansion of the exact solution about $x(t_m)$, we get that the exact solution for time t_{m+1} is given by

$$x^E(t_{m+1}) = x(t_m) + h\,\dot{x}\,(t_m) + O(h^2)$$

whereas the approximate solution computed from Eqn. (3.1) is

$$x(t_{m+1}) = x(t_m) + h\,\frac{x(\infty) - x(t_m)}{\tau} + O(h^2).$$

Therefore, for consistency, $x(\infty)$ and τ must satisfy the condition

$$\frac{x(\infty) - x(t_m)}{\tau} = \dot{x}\,(t_m). \tag{3.2}$$

Using the relation in Eqn. (3.2) we can rewrite Eqn. (3.1) as

$$x(t_{m+1}) = x(t_m) + \tau(1 - e^{-\frac{h}{\tau}})\,\dot{x}\,(t_m) \tag{3.3}$$

which defines a family of consistent one-step first-order explicit exponentially fitted formulas parameterized by τ.

Also note that when $h \to 0$

$$(1 - e^{-\frac{h}{\tau}}) \approx \frac{h}{\tau} \tag{3.4}$$

which implies that formula 3.3 reduces to the well known forward-Euler integration algorithm and therefore inherits its convergence properties. That is, when numerically integrating on a finite interval $[0, T]$

$$\lim_{h \to 0} \max_{0 < m < M} \|x(t_m) - x^E(t_m)\| = 0 \tag{3.5}$$

where $t_M = T$.

For a system of N ordinary differential equations, Eqn. (3.3) can be generalized as,

$$\mathbf{x}(t_{m+1}) = \mathbf{x}(t_m) + \mathbf{D}^{-1}(\mathbf{I} - e^{-h\mathbf{D}})\,\dot{\mathbf{x}}\,(t_m) \tag{3.6}$$

where $\mathbf{D}$ is a $N \times N$ diagonal matrix, with $d_i = \frac{1}{\tau_i}$, $i \in \{1, ..., n\}$.

Both of the approaches followed in the circuit simulators CINNAMON and XPSim can be reduced to the above formulation. It is in the choice of the fitting matrix $\mathbf{D}$ that the algorithms differ significantly. We will illustrate the difference between the two approaches for the matrix test problem

$$\dot{\mathbf{x}} = -\mathbf{A}\mathbf{x} \qquad \mathbf{x}(0) = \mathbf{x}_0 \neq \mathbf{0} \tag{3.7}$$

where $\mathbf{A} \in \Re^{n \times n}$. We will assume that $\mathbf{A}$ is connectedly diagonally-dominant [Varga62] with positive diagonals, as this models the equations resulting from the way MOS circuits are treated in timing simulation programs. The study of this simple linear time-invariant problem will allows to fully understand the exponential-fitting algorithms described.

In CINNAMON, a simple approach is used: the τ_i is selected from the diagonal term of the $\mathbf{A}$ matrix as

$$\tau_i = \frac{1}{a_{ii}} \tag{3.8}$$

and the $\mathbf{x}(\infty)$ is set according to the consistency relationship (3.2) to be:

$$x_i(\infty) = x_i(t_m) + \tau_i\,\dot{x}_i\,(t_m) \tag{3.9}$$

for $i \in \{1, ..., n\}$.

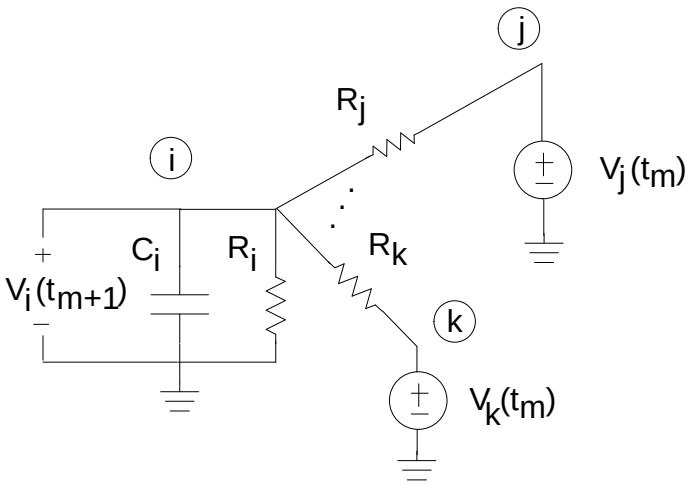

FIGURE 3-1: Updating a node value with CINNAMON (circuit interpretation)

There is a simple circuit interpretation of the CINNAMON algorithm. Each node in the circuit is updated to a new timepoint by computing the exact solution to that node, given all the other nodes are treated as fixed voltage sources with values given from the previous timepoint (see Figure (3-1)).

In the XPSim algorithm, the value selected for $\mathbf{x}(\infty)$ is, for a linear system, the *correct* steady-state value. Note that when applied to nonlinear circuits, the $\mathbf{x}(\infty)$ used is not necessarily the correct steady-state, and may change at each timestep, adding to the computational cost of the method. However, if the intention is to simulate digital logic circuitry, the $\mathbf{x}(\infty)$ computation can possibly be avoided by exploiting the fact that eventually the circuit voltages approach the power supply or ground. For our problem, $\mathbf{x}(\infty) = \mathbf{0}$, and by consistency (3.2) reduces to

$$\tau_i = -\frac{x_i(t_m)}{\dot{x}_i(t_m)} \tag{3.10}$$

and

$$x_i(t_{m+1}) = x_i(t_m) e^{h\frac{\dot{x}_i(t_m)}{x_i(t_m)}} = x_i(t_m) \exp\left[-h\frac{\sum_{j=1}^{n} a_{ij}x_j(t_m)}{x_i(t_m)}\right] \tag{3.11}$$

for $i \in \{1, ..., n\}$.

Clearly, the XPSim algorithm can not be used when the solution passes through zero. In that case the XPSim algorithm is not even consistent, as the method is positive invariant. That is, the XPSim algorithm can only produce a positive $x(t_{m+1})$ from a positive $x(t_m)$. For our examples, this difficulty can be mostly ignored if the initial condition is assumed to be a positive vector. Then, if the problem is as given in Eqn. (3.7), where $\mathbf{A}$ is strictly or connectedly diagonally dominant, the exact solution and the solution computed by XPSim will be positive for all time [Sandberg74].

3.2.2 Large Timestep Properties of Exponential Fitting Formulas

Because exponential-fitting methods are tuned to scalar problems, they obviously are going to perform well when **A** of Eqn. (3.7) is strongly diagonally dominant, but they degrade surprisingly quickly when **A** is only weakly diagonally dominant. Consider the example in Figure 3-2, a tightly coupled two-node circuit where the initial condition at each node is 1 volt.

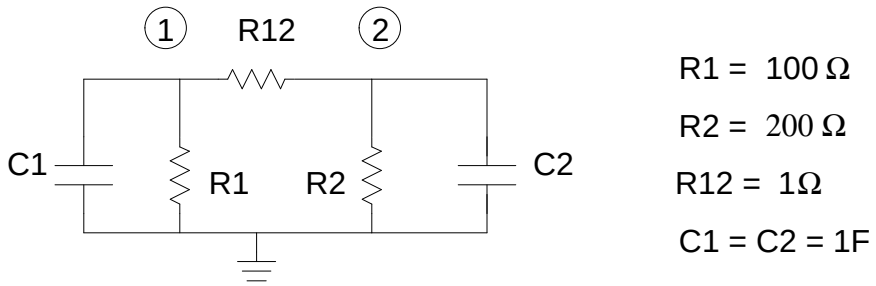

Figure 3-2: Test Circuit for Explicit Integration Formulas

A comparison of the computed results simulating this circuit using the backward-Euler (BE), defined in Eqn. (2.11), the XPSim (XP), and the CINNAMON (CIN) algorithms (another algorithm, (IP) also in this figure, is described below) with a 0.1 second timestep is plotted in Fig. 3-3. As is clear from the picture, all algorithms produce roughly the same accurate results.

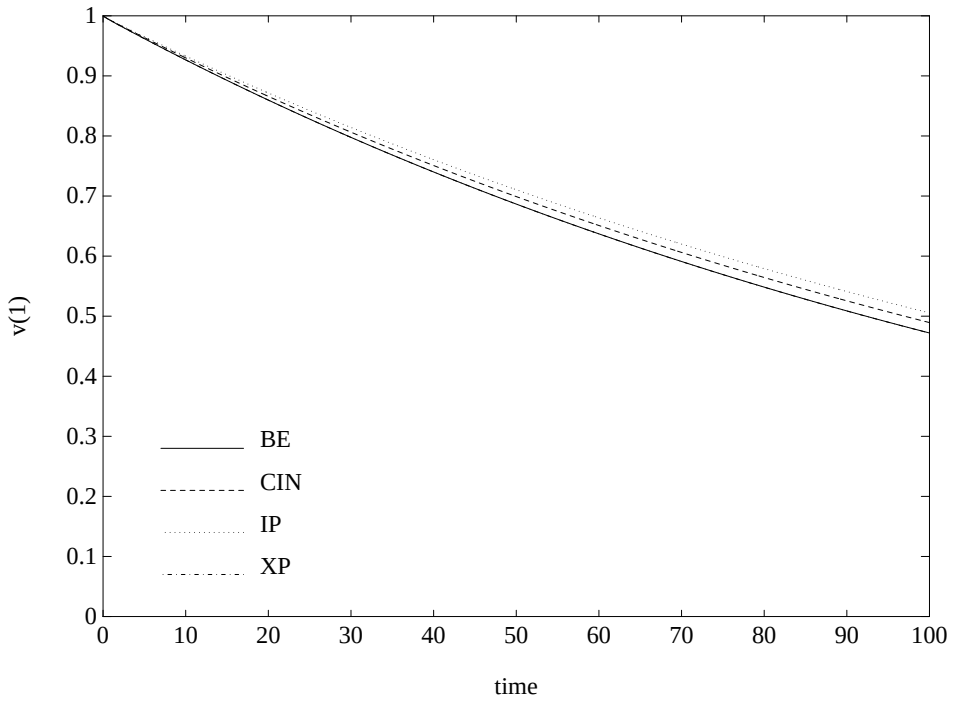

Figure 3-3: Voltage in node 1 for $h = 0.1$

In Fig. 3-4, the results from simulating the same example, but using a 1.1 second timestep,

are plotted. For this case, disappointing results are achieved for the CINNAMON (CIN) and XPSim (XP) methods when compared to the standard backward-Euler (BE) algorithm. The solution computed with XPSim is unstable and the solution computed with CINNAMON is inaccurate in that it decays too slowly.

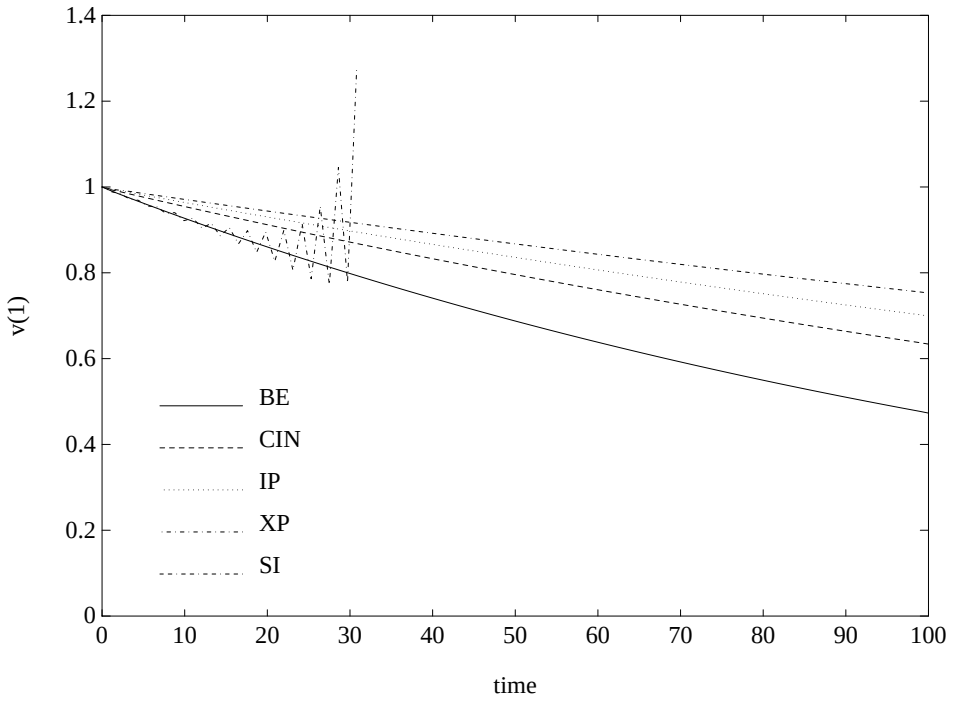

FIGURE 3-4: Voltage in node 1 for $h = 1.1$

If the timestep is set to 10 seconds, as in Fig. 3-5, the result from the XPSim algorithm becomes too unstable to plot, and the result from the CINNAMON algorithm gets "stuck", that is, it decays much more slowly than the exact solution (the algorithms (SI), (IP), also in these figures, are described below). This property of the CINNAMON algorithm, that of getting "stuck", is particularly insidious as the slow changing node may be misinterpreted as having achieved equilibrium. This may confuse an event-driven algorithm, and may lead to significant errors.

It is possible to stabilize the XPSim algorithm for diagonally dominant problems by making the computation semi-implicit. When applied to Eqn. (3.7), a semi-implicit version of XPSim, which we denote as IPSim is

$$x_i(t_{m+1}) = x_i(t_m) \exp\left[-h \frac{a_{ii} x_i(t_{m+1}) + \sum_{j \neq i} a_{ij} x_j(t_m)}{x_i(t_m)}\right] \tag{3.12}$$

for $i \in \{1, ..., n\}$. Note the method is not implicit with respect to the denominator term in the exponential, as this does not enhance stability and makes for a harder nonlinear problem to solve at each step. The IPSim algorithm is similar to the Gauss-Jacobi semi-implicit algorithm

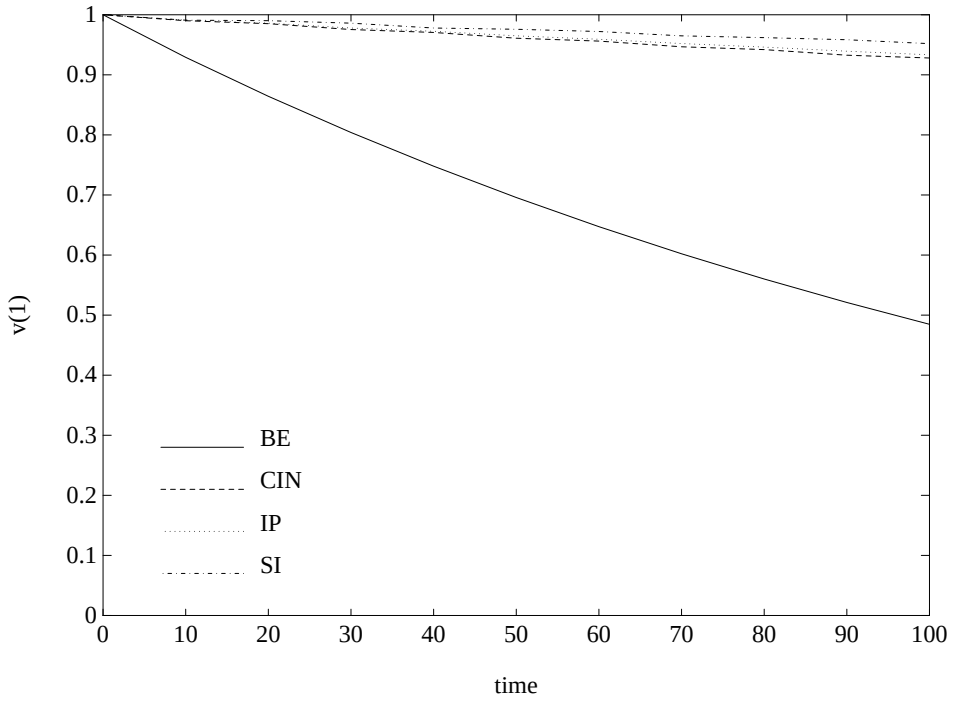

FIGURE 3-5: Voltage in node 1 for $h = 10$

used in the MOTIS [Chawla75] program, which for our test problem yields an update equation

$$x_i(t_{m+1}) = x_i(t_m) - h\left[a_{ii}x_i(t_{m+1}) + \sum_{j\neq i} a_{ij}x_j(t_m)\right] \tag{3.13}$$

for $i \in \{1, ..., n\}$.

It is known [White86] that the Gauss-Jacobi semi-implicit method is A-stable for $\mathbf{A}$ in Eqn. (3.7) connectedly diagonally dominant, and this result also holds for CINNAMON and IPSim, which we prove below. First we present a definition of connectedly diagonally dominant and state and prove a lemma that will be helpful to prove the stated properties of the CINNAMON algorithm. For completeness, we also present the usual definition of the L-infinity norm (L_∞) and the spectral radius, as we will use them throughout this thesis.

L-INFINITY NORM
Definition 3.1 *The* **L-infinity norm**, *denoted as* L_∞, *of a vector* $\mathbf{x} \in \Re^{n\times n}$ *is defined as* $\|\mathbf{x}\|_\infty \equiv max_{i\in\{1,...,n\}}|x_i|$.

SPECTRAL RADIUS

Definition 3.2 *Given a matrix* $\mathbf{A}$ *with eigenvalues* λ_i , $1 \leq i \leq n$, *then*

$$\rho(\mathbf{A}) \equiv max_{1 \leq i \leq n} |\lambda_i|$$

is the spectral radius of the matrix $\mathbf{A}$.

CONNECTEDLY DIAGONALLY DOMINANT MATRIX

Definition 3.3 *A matrix* $\mathbf{A}$ *is* **connectedly diagonally dominant** *if it is diagonally dominant, that is*

$$\forall i \in \{1, ..., n\} \; : \; |a_{ii}| \geq \sum_{j \neq i}^{n} |a_{ij}| \tag{3.14}$$

and for every row i *it is always possible to find a path to it of the form:*

$$a_{k_1 k_2} \rightarrow ... \rightarrow a_{k_{\nu-1} k_\nu} \rightarrow a_{k_\nu i}$$

where row k_1 *is a a strictly diagonally dominant row, i.e., one for which the inequality in Eqn. (3.14) is strict.*

There is a clear interpretation for connectedly diagonally dominant matrices in the context of linear resistive circuits. A circuit for which there is some path of resistors from every node to ground will have a connectedly diagonally dominant conductance matrix.

Lemma 3.1 *Given* $\mathrm{M} = \mathcal{F}(\mathbf{A})$ *such that*

$$\begin{aligned} m_{ii} &= g(a_{ii}) \\ m_{ij} &= -\frac{a_{ij}}{f(a_{ii})} \qquad\qquad i \neq j \end{aligned} \tag{3.15}$$

for $i, j \in \{1, ..., n\}$, *where the functions* $f(a_{ii})$ *and* $g(a_{ii})$ *satisfy:*

$$\begin{aligned} &(i) \qquad f(a_{ii}) > a_{ii} \\ &(ii) \qquad g(a_{ii}) \leq 1 - \tfrac{a_{ii}}{f(a_{ii})} \end{aligned}$$

Then if $\mathbf{A}$ *is connectedly diagonally dominant with* $a_{ii} > 0$, $\forall i$, $\rho(\mathrm{M}) < 1$.

Proof: In general

$$\rho(\mathrm{M}) \leq max_i \{ \sum_{j=1}^{n} |m_{ij}| \}$$

from Eqns. (3.15)

$$\rho(\mathbf{M}) \leq max_i\{\sum_{j=1}^{n} |m_{ij}|\} = max_i\{g(a_{ii}) + \frac{\sum_{j=1,j\neq i}^{n} |a_{ij}|}{f(a_{ii})}\} \tag{3.16}$$

Since $\mathbf{A}$ is connectedly diagonally dominant, it follows from Eqn. (3.16)

$$max_i\{g(a_{ii}) + \frac{\sum_{j=1,j\neq i}^{n} |a_{ij}|}{f(a_{ii})}\} \leq max_i\{g(a_{ii}) + \frac{a_{ii}}{f(a_{ii})}\} \leq 1 \tag{3.17}$$

If $\mathbf{A}$ is strictly diagonally dominant then the first inequality of Eqn. (3.17) is strict, since $a_{ii} > 0$ and $a_{ii} > \sum_{j\neq i}^{n} |a_{ij}|$, and therefore $\rho(\mathbf{M}) < 1$ which proves the Lemma.
If $\mathbf{A}$ is not strictly diagonally dominant, but only connectedly diagonally dominant, then Eqns. (3.16) and (3.17) only imply $\rho(\mathbf{M}) \leq 1$. Suppose that $\rho(\mathbf{M}) = 1$. Then there exists an eigenvalue of $\mathbf{M}$, λ, such that $|\lambda| = 1$. If $\mathbf{v}$ is the eigenvector associated with that eigenvalue, by definition,

$$\forall k \in \{1, ..., n\} \; : \; m_{kk}v_k + \sum_{j\neq k, j=1}^{n} m_{kj}v_j = \lambda v_k \tag{3.18}$$

The eigenvector $\mathbf{v}$ must have some nonzero maximum element,

$$\exists i \in \{1, ..., n\} \; : \; |v_i| \geq |v_j| \qquad \forall j \neq i \quad and \quad |v_i| > 0 \tag{3.19}$$

Dividing through by v_i in Eqn. (3.18), for k selected equal to i, taking absolute values, and using the fact that $\lambda = 1$, leads to

$$|m_{ii}| + \sum_{j\neq i, j=1}^{n} |m_{ij}||\frac{v_j}{v_i}| \geq 1 \tag{3.20}$$

From Eqn. (3.17)

$$\sum_{j=1}^{n} |m_{ij}| \leq 1 \tag{3.21}$$

If the sum of Eqn. (3.21) is strictly less than one, which would be the case if the i^{th} row of $\mathbf{A}$ were strictly diagonally dominant, then Eqn. (3.20) could not be true and therefore $\rho(\mathbf{M}) < 1$ proving the Lemma. If the sum of Eqn. (3.21) is precisely one, then Eqn. (3.20) could be true only if $|\frac{v_j}{v_i}| = 1$ for all j for which $m_{ij} \neq 0$. If $|\frac{v_j}{v_i}| = 1$ then $|v_j| > 0$ thus implying that Eqn. (3.20) could also be written for any row j for which $m_{ij} \neq 0$. If for any of these j rows $\mathbf{A}$ is strictly diagonally dominant in that row, then again Eqn. (3.20) cannot be true and therefore $\rho(\mathbf{M}) < 1$. Continuing this process, since by hypothesis $\mathbf{A}$ is connectedly diagonally dominant, eventually Eqn. (3.20) will be written for some row k that is a strictly diagonally dominant row of $\mathbf{A}$, and this will finally imply that $\rho(\mathbf{M}) < 1$. ∎

CINNAMON A-STABILITY

Theorem 3.1 *When applied to solving Eqn. (3.7), where* $\mathbf{A}$ *is assumed to be strictly or connectedly diagonally dominant, CINNAMON is A-stable.*

Proof: The CINNAMON algorithm for updating x can be written in the form $\mathbf{x}(t_{m+1}) = \mathbf{M}\mathbf{x}(t_m)$ where $\mathbf{M}$ is given by

$$\mathbf{M} = \mathbf{I} - \mathbf{D}^{-1}(\mathbf{I} - e^{-h\mathbf{D}})\mathbf{A}.$$

Note that the individual entries of $\mathbf{M}$ are given by

$$\begin{aligned} m_{ii} &= e^{-ha_{ii}} \\ m_{ij} &= -(1 - e^{-ha_{ii}})\frac{a_{ij}}{a_{ii}} \qquad i \neq j \end{aligned}$$

It follows then that the CINNAMON algorithm is A-stable for our test problem if and only if the eigenvalues of $\mathbf{M}$ are inside the unit circle for all $h > 0$. That this is true can be shown noting that

$$m_{ij} = -(1 - e^{-ha_{ii}})\frac{a_{ij}}{a_{ii}} = -\frac{a_{ij}}{f(a_{ii})} \qquad \forall j \neq i$$

where $f(a_{ii}) = \frac{a_{ii}}{1-e^{-ha_{ii}}} > a_{ii}$ for $a_{ii} > 0$ and

$$m_{ii} = e^{-ha_{ii}} = g(a_{ii}) = 1 - \frac{a_{ii}}{f(a_{ii})}$$

and then applying Lemma 3.1. ∎

IPSIM A-STABILITY

Theorem 3.2 *When applied to solving Eqn. (3.7), where* $\mathbf{A}$ *is assumed to be strictly or connectedly diagonally dominant, and the initial condition is a positive vector, IPSim is A-stable.*

In order to prove the A-stability of the semi-implicit IPSim method, we first prove some useful lemmas.

Lemma 3.2 *When applied to solving Eqn. (3.7), where* $\mathbf{A}$ *is assumed to be strictly or connectedly diagonally dominant, and the initial condition is a positive vector, the* L_∞ *norm of the updated vector computed using IPSim cannot increase, that is,*

$$\forall l \in \mathcal{N}, l \geq 1 \ : \ \|\mathbf{x}(t_{m+l})\|_\infty \leq \|\mathbf{x}(t_m)\|_\infty \tag{3.22}$$

Proof: The positive initial condition implies that for all j, $x_j(t_m)$ is positive and therefore to prove the lemma it is sufficient to show that $|x_j(t_{m+1})| \leq \|\mathbf{x}(t_m)\|_\infty$ for all j and m. In general, the IPSim update equation for x_i at time t_m can be written as

$$x_i(t_{m+1}) = x_i(t_m) \exp\left[-h\frac{a_{ii}x_i(t_{m+1}) + \sum_{j\neq i} a_{ij}x_j(t_m)}{x_i(t_m)}\right]. \tag{3.23}$$

for $i \in \{1, ..., n\}$. Suppose by contradiction, that

$$\exists j \ : \ x_j(t_{m+1}) > \|\mathbf{x}(t_m)\|_\infty. \tag{3.24}$$

Then

$$a_{jj}x_j(t_{m+1}) + \sum_{k\neq j} a_{kj}x_k(t_m) > 0 \tag{3.25}$$

as $\mathbf{A}$ is assumed to be diagonally dominant and $a_{jj} > 0$. But then, from Eqn. (3.23) $x_j(t_{m+1}) < x_j(t_m) \leq \|\mathbf{x}(t_m)\|_\infty$ which contradicts Eqn (3.24). ∎

Lemma 3.3 *If a row i of $\mathbf{A}$ is strictly diagonally dominant with a positive diagonal, that is if*

$$1 - \sum_{j\neq i, j=1}^{n} \frac{|a_{ij}|}{a_{ii}} \equiv \epsilon_i > 0, \tag{3.26}$$

for $i \in \{1, ..., n\}$, then given $\mathbf{x}(t_0)$ is a positive vector, then, for all $h > 0$, there is a $\gamma_i < 1$, independent of $\mathbf{x}$ and m, for which $x_i(t_{m+1})$ produced by the IPSim algorithm satisfies

$$x_i(t_{m+1}) \leq \gamma_i \|\mathbf{x}(t_m)\|_\infty. \tag{3.27}$$

Proof: As $x_i(t_m)$ is positive, given $\mathbf{x}(t_0)$ is a positive vector, Eqn. (3.27) is violated only if

$$x_i(t_{m+1}) > \gamma_i \|\mathbf{x}(t_m)\|_\infty. \tag{3.28}$$

To establish a contradiction to prove this Lemma, we start by assuming Eqn. (3.28) is true. Applying this assumption to Eqn. (3.12), and using $x_i(t_m) \leq \|\mathbf{x}(t_m)\|_\infty$ by definition, leads to the following bound on $x_i(t_{m+1})$:

$$x_i(t_{m+1}) \leq \|\mathbf{x}(t_m)\|_\infty \exp\left[-h\frac{(\gamma_i a_{ii} - \sum_{i\neq j} |a_{ij}|)\|\mathbf{x}(t_m)\|_\infty}{x_i(t_m)}\right]. \tag{3.29}$$

Substituting the definition of ϵ_i from Eqn. (3.26) leads to

$$x_i(t_{m+1}) \leq \|\mathbf{x}(t_m)\|_\infty \exp\left[-h\frac{(\epsilon_i + \gamma_i - 1)a_{ii}\|\mathbf{x}(t_m)\|_\infty}{x_i(t_m)}\right]. \tag{3.30}$$

In order to prove this Lemma, it is sufficient to consider values for γ in the range $[1-\epsilon_i, 1]$, as $\epsilon_i > 0$. For this range of values, $(\epsilon_i + \gamma - 1)$ is positive and therefore

$$x_i(t_{m+1}) \leq \|\mathbf{x}(t_m)\|_\infty \exp\left[-h\frac{(\epsilon_i+\gamma-1)a_{ii}\|\mathbf{x}(t_m)\|_\infty}{\|\mathbf{x}(t_m)\|_\infty}\right] \tag{3.31}$$

which simplifies to

$$x_i(t_{m+1}) \leq \|\mathbf{x}(t_m)\|_\infty \exp\left[-h(\epsilon_i+\gamma-1)a_{ii}\right] \tag{3.32}$$

The term $\exp\left[-h(\epsilon_i+\gamma-1)a_{ii}\right]$ is a continuous function of γ whose value is 1 when $\gamma = 1-\epsilon_i$ and whose value is $e^{-h\epsilon_i a_{ii}} < 1$ when $\gamma = 1$. It follows then that there exists a $\tilde{\gamma}_i < 1$ such that

$$\tilde{\gamma}_i = \exp\left[-h(\epsilon_i+\tilde{\gamma}_i-1)a_{ii}\right]. \tag{3.33}$$

For this $\tilde{\gamma}_i < 1$,

$$x_i(t_{m+1}) \leq \tilde{\gamma}_i\|\mathbf{x}(t_m)\|_\infty \tag{3.34}$$

which contradicts Eqn. (3.28) for the case $\gamma_i = \tilde{\gamma}_i$ and proves the Lemma. ∎

Lemma 3.4 *Given a row i of $\mathbf{A}$ which is diagonally dominant with a positive diagonal, if there is a j for which $a_{ij} \neq 0$ and the sequence produced by the IPSim algorithm applied to Eqn. (3.7) with a positive vector initial condition is such that for all $h > 0$,*

$$x_j(t_{m+1}) \leq \gamma_j\|\mathbf{x}(t_m)\|_\infty \tag{3.35}$$

for some $\gamma_j < 1$ independent of $\mathbf{x}$ and m, then for all $h > 0$,

$$x_i(t_{m+2}) \leq \gamma_i\|\mathbf{x}(t_m)\|_\infty \tag{3.36}$$

for some $\gamma_i < 1$, independent of $\mathbf{x}$ and m.

Proof: As in Lemma 3.3, we start by assuming

$$x_i(t_{m+2}) > \gamma_i\|\mathbf{x}(t_m)\|_\infty \tag{3.37}$$

to show a contradiction. Following steps similar to those in Lemma 3.3, $x_i(t_{m+2})$ can be shown to be bounded by

$$x_i(t_{m+2}) \leq \|\mathbf{x}(t_m)\|_\infty \exp\left[-h(a_{ii}(\gamma_i-1) + |a_{ij}|(1-\gamma_j))\right] \tag{3.38}$$

where γ_i is assumed to be in the interval $\left[1-\frac{|a_{ij}|}{a_{ii}}(1-\gamma_j), 1\right]$. Again applying arguments similar to those in Lemma 3.3, we arrive at the conclusion that there exists a $\tilde{\gamma}_i < 1$ such that

$$x_i(t_{m+2}) \leq \tilde{\gamma}_i\|\mathbf{x}(t_m)\|_\infty \tag{3.39}$$

which contradicts Eqn. (3.28) for the case $\gamma_i = \tilde{\gamma}_i$ and proves the Lemma. ∎

Proof of Theorem 3.2: We show that the IPSim algorithm is A-stable, given the theorem conditions above, by proving that they imply

$$\|\mathbf{x}(t_{m+N})\|_\infty \leq \gamma \, \|\mathbf{x}(t_m)\|_\infty \tag{3.40}$$

for some $N < n$, where for all $h > 0$, $\gamma < 1$ and independent of $\mathbf{x}$ and m. That A-stability follows from Eqn. (3.40) can be seen by considering that Eqn. (3.40) implies

$$\lim_{k \to \infty} \|\mathbf{x}(t_{m+kN})\|_\infty = 0. \tag{3.41}$$

Then combining Eqn. (3.41) with the result of Lemma 3.2 yields

$$\lim_{l \to \infty} \|\mathbf{x}(t_{m+l})\|_\infty = 0 \tag{3.42}$$

for all $h > 0$, demonstrating A-stability.

To prove Eqn. (3.40) we use the results of Lemmas 3.3 and 3.4. From Lemma 3.3,

$$x_i(t_{m+1}) \leq \gamma_i \|\mathbf{x}(t_m)\|_\infty.$$

for all i associated with rows of $\mathbf{A}$ which are strictly diagonally dominant. Lemma 3.4 then implies that for all rows j of $\mathbf{A}$ which are connected to strictly diagonally dominant rows i,

$$x_j(t_{m+2}) \leq \gamma_j \|\mathbf{x}(t_m)\|_\infty$$

where again $\gamma_j < 1$. Also, by Lemma 3.2

$$x_i(t_{m+2}) \leq x_i(t_{m+1}) \leq \gamma_i \|\mathbf{x}(t_m)\|_\infty.$$

By induction, and given that $\mathbf{A}$ is connectedly diagonally dominant,

$$x_k(t_{m+N}) \leq \gamma_k \|\mathbf{x}(t_m)\|_\infty \tag{3.43}$$

for all $k \in \{1, ..., n\}$ and $\gamma_k < 1$. Here N is the length of the longest path from any row to a strictly diagonally dominant row, or in a connected circuit, the length of the longest path from any node to ground. It then follows that Eqn. (3.43) implies Eqn. (3.40) where $\gamma = max\{\gamma_1, ..., \gamma_n\} < 1$. ∎

The results using the IPSim (IP) and MOTIS (SI) algorithms to simulate the circuit in Fig. 3-2 with 1.01 and 10 second timesteps are plotted in Fig. 3-4 and Fig. 3-5 respectively. As the plots show, the IPSim algorithm is stable, but produces results not significantly more accurate than the MOTIS or CINNAMON algorithms. We make this statement more rigorous in the following theorem whose proof ends this section.

LARGE TIMESTEP BEHAVIOR

Theorem 3.3 *If* **A** *has positive diagonal entries, then in the limit as the timesteps becomes large, the CINNAMON, IPSim, and MOTIS algorithms produce identical results.*

Proof: Summarizing, the update equations for the MOTIS, CINNAMON, and IPSim algorithms are respectively:

$$x_i(t_{m+1}) = x_i(t_m) - h\left[a_{ii}x_i(t_{m+1}) + \sum_{j\neq i} a_{ij}x_j(t_m)\right], \tag{3.44}$$

$$x_i(t_{m+1}) = x_i(t_m) - \frac{1-e^{-ha_{ii}}}{a_{ii}}\left[\sum_{j=1}^{n} a_{ij}x_j(t_m)\right], \tag{3.45}$$

$$x_i(t_{m+1}) = x_i(t_m)\exp\left[-h\frac{a_{ii}x_i(t_{m+1}) + \sum_{j\neq i} a_{ij}x_j(t_m)}{x_i(t_m)}\right]. \tag{3.46}$$

It is easy to see that in the limit of large h, and given that a_{ii} is positive for all i, that $x(t_{m+1})$ for both the MOTIS and CINNAMON algorithms is

$$x_i(t_{m+1}) = -\sum_{j\neq i}\frac{a_{ij}}{a_{ii}}x_j(t_m). \tag{3.47}$$

The result in Eqn. (3.47) holds true for the IPSim algorithm as well. This can be seen by considering that in the limit of large h, the argument of the exponential in IPSim's update equation (3.46) cannot go to ∞. Therefore, the term from the numerator of the exponential's argument in Eqn. (3.46)

$$a_{ii}x_i(t_{m+1}) + \sum_{j\neq i} a_{ij}x_j(t_m) \tag{3.48}$$

must approach zero as $h \to \infty$. ∎

In fact the result produced by all the algorithm is the solution of the one node circuit depicted in Figure 3-6.

3.2.3 Effects of Equation Ordering

In general, it is possible to improve the stability and accuracy of explicit or semi-implicit methods by ordering the equations being solved and using updated values when possible. Specifically, when calculating $x_i(t_{m+1})$ the use of the already computed values of $x_j(t_{m+1})$ for $j < i$ will improve the accuracy of the solution. This is exploited in most implementations of semi-implicit integration algorithms [Saleh83] [Chen84], and also in the CINNAMON circuit simulator [Neto89]. Ordering is in effect a serialization of the computations performed by the algorithm. Ordering had already been mentioned when the Gauss-Jacobi and Gauss-Seidel algorithms were introduced in chapter 2. It can be proved that ordered algorithms, although

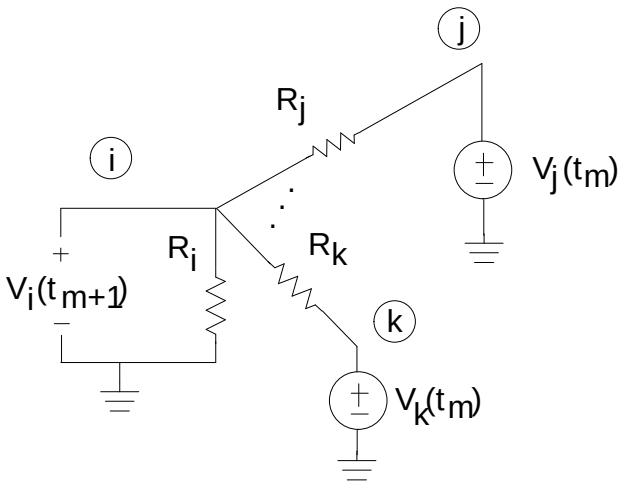

FIGURE 3-6: Circuit used for determining $\mathbf{x}_\infty$ for CINNAMON, XPSim and SI

presenting advantageous properties in terms of stability and accuracy (or rate of convergence in the relaxation case) are to some extent less parallelizable than their unordered counterparts. Nevertheless, if a simple ordering can effectively increase the accuracy of an integration formula, the decrease in exploitable parallelism may still be compensated and therefore worth studying.

Using ordering in the XPSim algorithm leads to the following update equation for x_i:

$$x_i(t_{m+1}) = x_i(t_m) \exp\left[-h\frac{\sum_{j<i} a_{ij}x_j(t_{m+1}) + \sum_{j\geq i} a_{ij}x_j(t_m)}{x_i(t_m)}\right], \tag{3.49}$$

where subscript index also indicates the ordering.

In Figure 3-7 we present the plots of computed solutions to the test circuit in Figure 3-2, under the same conditions of Figure 3-4, i.e., using a 1.1 second timestep. The waveforms shown correspond to backward-Euler (BE) and ordered versions of CINNAMON (CINo), IPSim (IPo), XPSim (XPo), and the semi-implicit algorithm (SIo). Here, the ordering used always updates node 1 first. From the figure, and comparing to the results shown in Figure 3-4, one can see that ordering improved the accuracy of all methods, most notably that of XPSim which is stabilized by ordering. Also from the plot we note that with ordering, the solution produced with CINNAMON becomes quite accurate, and that the solution produced with IPSim is almost the same as that obtained with the semi-implicit algorithm, albeit at the expense of more computation.

The results of the above plot are partially deceptive in that it is not always possible to stabilize the XPSim algorithm by ordering, as can be demonstrated by considering the example circuit in Figure 3-8. Solutions for the initial condition of both nodes at $1V$ were computed using node 1 first ordered XPSim (XPo), node 1 first ordered CINNAMON (CINo), and backward-Euler (BE), all with a timestep $h = 6$, and are plotted in Figure 3-9. Note that the solution computed with the ordered XPSim method shows oscillations of increasing magnitude. This plot is somewhat surprising since the system matrix is not far from diagonal. In addition, if the

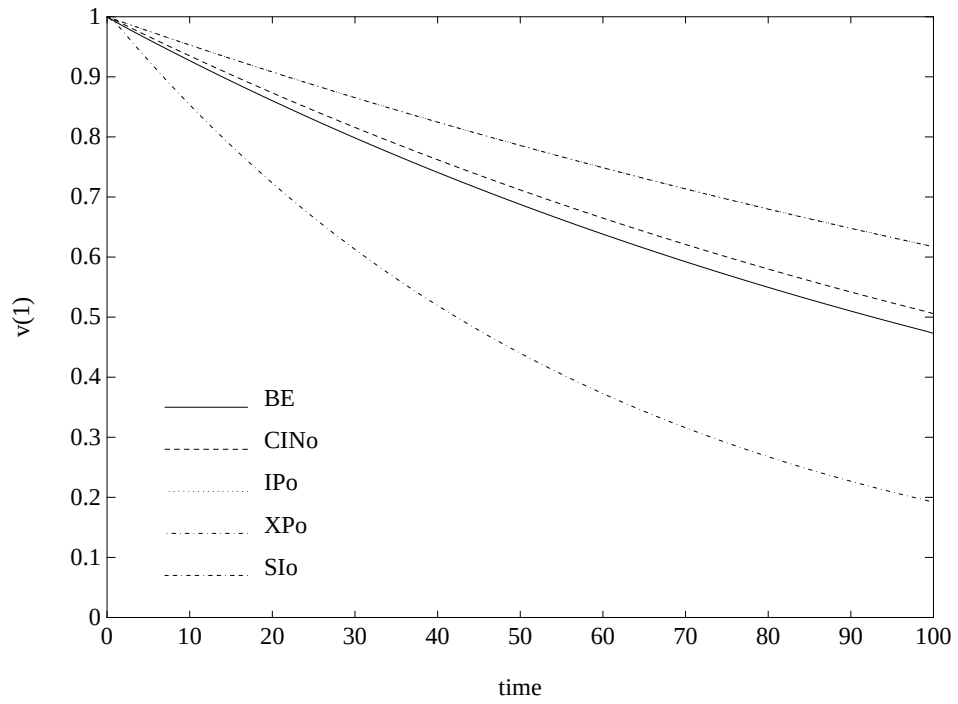

FIGURE 3-7: Voltage in node 1 for $h = 1.1$

timestep is made larger, to $h = 7$, then it is *impossible* to stabilize XPSim with any ordering, a fact that can be established by exhaustive search over all the possible orderings.

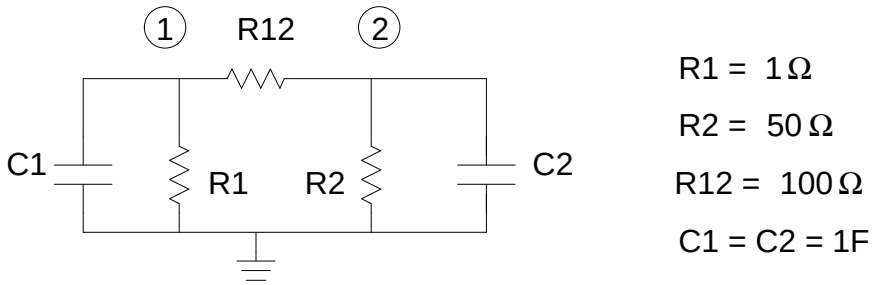

FIGURE 3-8: Circuit to test possible stability of the ordered XPSim algorithm

Even though ordering is not guaranteed to stabilize the XPSim algorithm, frequently it does, if a good ordering is selected. We have found the following ordering heuristic to be helpful:

Heuristic 3.1 *Updating the nodes in order of increasing derivative, starting with the node associated with the most negative derivative, is referred to as the fastest-first ordering heuristic.*

The *fastest-first* ordering heuristic seems very natural in the sense that updating the node with the most negative derivative is a greedy attempt to force as much a decrease as possible.

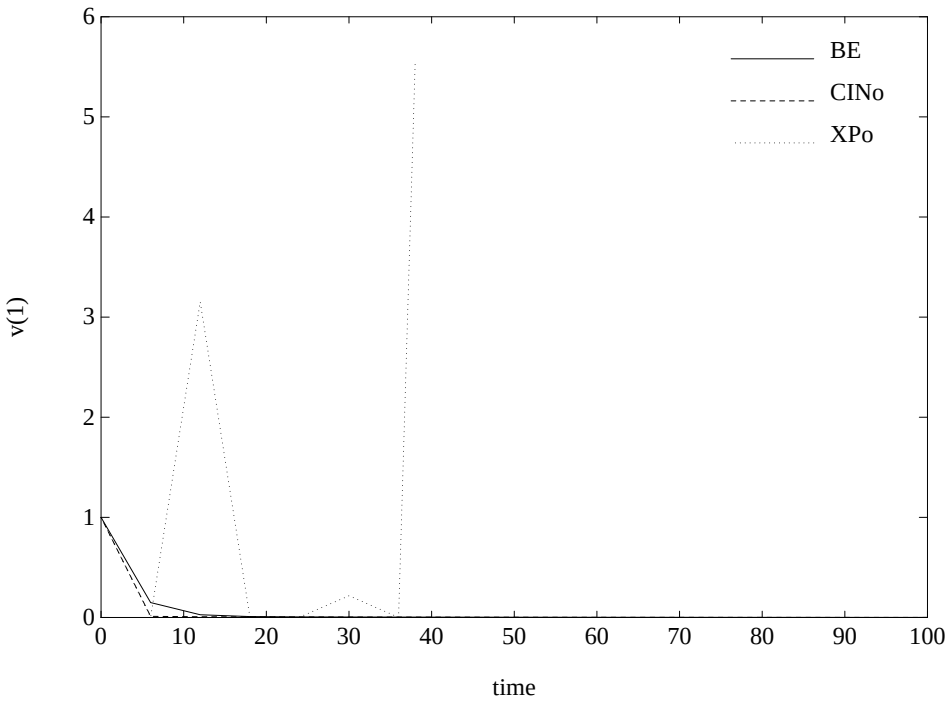

FIGURE 3-9: $v_1(t)$ from the circuit in Figure 3-8 for $h = 6$

The method clearly only makes sense in the case of solutions decaying to zero. To show that this ordering is effective for both CINNAMON and XPSim style algorithms, the circuit in Figure 3-2 was simulated using the initial conditions on the node voltage of $v_1(0) = 0.8V, v_2(0) = 1.0V$. The plots in Figure 3-10 show the waveforms obtained with backward-Euler (BE), the node 1 first ordered CINNAMON algorithm (CINf1), and both CINNAMON (CINff) and XPSim (XPff) using a *fastest-first* ordering scheme. A node 1 first ordered XPSim result is not plotted because it is violently unstable. This stems from the fact that node 1 is initially rising, which implies that the exponential XPSim must fit to has a positive argument, and numerical overflow occurs for a large enough timestep. It is worth noting that the CINNAMON style algorithms do not have this difficulty because for the test problem considered in this section, they are forced to fit to an exponential with a negative argument. From the plots we also note that the ordered XPSim is stabilized by being forced to move down much faster than the exact solution.

3.2.4 Discussion of Advantages and Drawbacks

With this study, some of the theoretical and practical aspects of using exponential-fitting integration formulas for MOS digital circuit timing analysis were clarified. Their stability properties have been properly studied and a unifying relation was established between several of the algorithms used in the past.

For the class of test problems studied, the CINNAMON, IPSim and semi-implicit are all A-

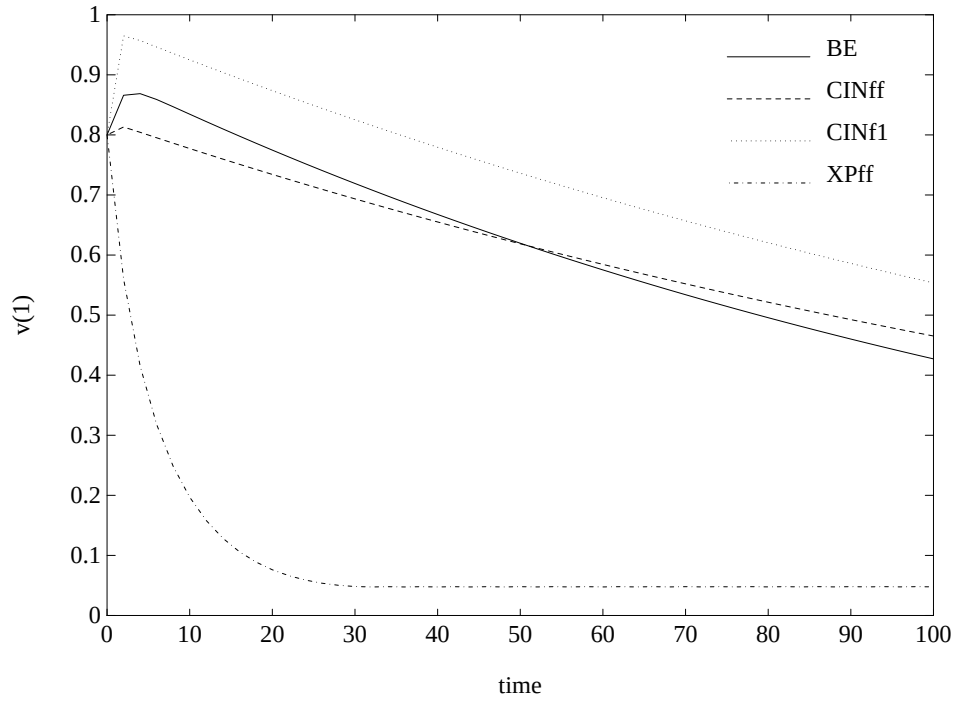

FIGURE 3-10: $v_1(t)$ from the circuit in Figure 3-2 for $h = 2, (v_1(0) = 0.8V)$

stable, and also all tend to get "stuck" for large timesteps, whereas the XPSim derived methods are generally unstable for large timesteps. In either case, ordering improves the situation. From our results in the previous section, we have seen that none of the methods are more accurate than backward-Euler for large timesteps and nondiagonal problems, although the methods may prove to be computationally less expensive.

If stability and accuracy over large timesteps is, as is usually true, the major concern, then the CINNAMON algorithm is probably superior to XPSim. The property of the CINNAMON style methods to get "stuck" can be considered a large drawback depending on the simulation control mechanism used. The XPSim style algorithms, which never get "stuck", have however a limited range of stability which will enforce smaller timesteps, and consequently more computation.

For the problem of general circuit simulation in the form envisioned in this study, we feel that these methods are of no great advantage. For once their accuracy is very reduced, which will imply that very small time steps will be required for the simulation process. With the exception of the CINNAMON algorithm, whose stability properties seem remarkable for an explicit algorithm, this accuracy does not get traded by stability, and even in CINNAMON this trade-off is not necessarily advantageous. As for the XPSim algorithm our results indicate that it presents no advantage over the forward-Euler algorithm and we therefore see no reason to investigate it further.

Ordering the equations was shown to improve the accuracy of all methods. In general

however, the increase in accuracy obtained from ordering the equations is accompanied by a decrease in exploitable parallelism, since ordering implies serialization of the computations. Furthermore, from our study, we have seen that the ordering heuristic producing requires substantial effort, both in terms of computation and also comparisons. Translated into the parallel processing world, this will in fact imply that many permutations of values, i.e. communication steps, must be performed at each step to determine the ideal ordering.

Based on this facts, these methods were not further investigated since it is our belief that they are not of general use for accurate electrical circuit simulation. In the rest of the work, we followed the more traditional approach of using implicit linear multistep formulas for discretizing the circuit differential equations.

3.3 Multiple Timepoint Algorithms

Multiple timepoint algorithms are studied here as possible candidates for circuit simulation. These type of algorithms present the interesting advantage from our point of view that they in fact increase the parallelism exploitable in the simulation problem.

Our goal in studying multiple timepoints is threefold:

1. maximize the efficiency of our implementation by achieving a good load balancing,
2. decrease interprocessor communication,
3. increase parallelism

Load balancing is an important requirement since we want to make the most use out of the resources made available. Interprocessor communication has to be minimized since we know it to be an expensive task. More parallelism is necessary so that larger speedups are provided even for moderately large circuits and also to provide an even usage of all processors.

To motivate the relevance of using multiple timepoint algorithms we consider the linear problem introduced in Eqn. (3.7) and repeated here for simplicity in Eqn. (3.50).

$$\frac{d}{dt}\mathbf{x} = -\mathbf{A}\mathbf{x} \qquad \mathbf{x}(0) = \mathbf{x}_0 \neq \mathbf{0} \qquad t \in [0, T] \tag{3.50}$$

Using the trapezoidal method, yields:

$$\left(\mathbf{I} + \frac{h}{2}\mathbf{A}\right)\mathbf{x}(t_{m+1}) = \left(\mathbf{I} - \frac{h}{2}\mathbf{A}\right)\mathbf{x}(t_m) \tag{3.51}$$

Given $\mathbf{x}(t_m)$ we can calculate $\mathbf{x}(t_{m+1})$ from Eqn. (3.51) and continue thereon for all $t \in [0, T]$

However, let us instead rewrite the above equation simultaneously for H timepoints, assuming for simplicity that fixed timesteps h are being used. This corresponds to writing the computations required to evaluate the value of $\mathbf{x}$ at H successive points in time and can be written in a larger matrix format as in Eqn. (3.52). The size of this matrix is $n \times H$, where

N is the size of the original problem and H is the number of timepoints at which the solution is guessed simultaneously in the given time interval. Such matrices have a form which, though dependent upon the multistep integration formula used, is always block-band-lower-triangular.

$$\begin{bmatrix} \mathbf{A}_1 & 0 & \cdots & \cdots & \cdots & 0 \\ \mathbf{B}_2 & \mathbf{A}_2 & \cdots & \cdots & \cdots & 0 \\ 0 & \cdots & \cdots & \cdots & \cdots & 0 \\ 0 & \cdots & \mathbf{B}_j & \mathbf{A}_j & \cdots & 0 \\ 0 & \cdots & \cdots & \cdots & \cdots & 0 \\ 0 & \cdots & \cdots & \cdots & \mathbf{B}_H & \mathbf{A}_H \end{bmatrix} \begin{bmatrix} \mathbf{x}(h) \\ \mathbf{x}(2h) \\ \vdots \\ \mathbf{x}(jh) \\ \vdots \\ \mathbf{x}(Hh) \end{bmatrix} = \begin{bmatrix} \mathbf{C}_1 \\ \mathbf{C}_2 \\ \vdots \\ \mathbf{C}_j \\ \vdots \\ \mathbf{C}_H \end{bmatrix} \tag{3.52}$$

Where the matrices $\mathbf{A}$, $\mathbf{B}$ and $\mathbf{C}$ are defined as follows.

$$\mathbf{A}_j = \left(\mathbf{I} + \frac{h}{2}\mathbf{A}\right) \qquad \forall j \tag{3.53}$$

$$\mathbf{B}_j = -\left(\mathbf{I} - \frac{h}{2}\mathbf{A}\right) \qquad \forall j \tag{3.54}$$

$$\mathbf{C}_1 = \left(\mathbf{I} - \frac{h}{2}\mathbf{A}\right)\mathbf{x}_0 \qquad \mathbf{C}_j = \mathbf{0} \quad \forall j > 1 \tag{3.55}$$

The advantages of using this type of algorithm become obvious now. For one, a larger matrix can be calculated and stored in parallel (even with non-constant terms) therefore utilizing more resources simultaneously. It is known that by using parallel cyclic reduction or nested dissection [Duff86], $\mathbf{M}$ can be factored in order $O(\log n)$ steps on a parallel processor with order $O(n)$ processors [Dongarra87]. This will be detailed further in section 5.2.2. Clearly, with the approach above, the size of the problem and the parallelism available have been increased.

Some of the questions arising with this type of algorithms can now be seen. Among them we note the pertinent point to be the placement of the timepoints and how to extrapolate the above derivation to nonlinear system.

3.3.1 The Waveform Newton Algorithm

The first of the two question in the end of the previous section is nontrivial and we will come back to it later. As for the second question, we will illustrate our approach by considering the following nonlinear problem in normal form:

$$\frac{d}{dt}\mathbf{x}(t) = \mathbf{f}(\mathbf{x}(t)) \qquad \mathbf{x}(0) = \mathbf{x}_0 \qquad t \in [0, T], \tag{3.56}$$

where $\mathbf{x}(t) \in \Re^n$ is the vector of unknowns and $\mathbf{f}(\mathbf{x}(t)) \in \Re^n$ is a nonlinear function of $\mathbf{x}(t)$.

Our approach is based on the Waveform-Newton algorithm [Kantorovich64, White85]. The Waveform-Newton algorithm uses a linearization of the problem around a guessed waveform on a certain time interval. The Waveform-Newton algorithm is readily derived, by considering the system of differential equations that describe the behavior of the system. To derive the

algorithm we cast the problem into that of finding a solution to Eqn. (3.56), which can be thought of as finding an $\mathbf{x}$ such that

$$\mathbf{F}(\mathbf{x}) = \frac{d}{dt}\mathbf{x} - \mathbf{f}(\mathbf{x}) = \mathbf{0}. \tag{3.57}$$

where $\mathbf{x} \in \omega$, the space of continuous and differentiable functions that map the interval $[0, T]$ into $\Re^n$, $\mathbf{F}$ takes $\omega \rightarrow \omega$, and $\mathbf{x}(0) = \mathbf{x}_0$. A pointwise description of $\mathbf{F}(\cdot)$ in t is given by

$$(\mathbf{F}(\mathbf{x}))(t) = \frac{d}{dt}\mathbf{x}(t) - \mathbf{f}(\mathbf{x}(t)). \tag{3.58}$$

The Newton-Raphson iteration for finding a solution to Eqn. (3.57) is,

$$\mathbf{J}_F(\mathbf{x}^k)\left[\mathbf{x}^{k+1} - \mathbf{x}^k\right] = -\mathbf{F}(\mathbf{x}^k) \tag{3.59}$$

where $\mathbf{J}_F(\mathbf{x})$ is the Frechet derivative of $\mathbf{F}(\mathbf{x})$ with respect to $\mathbf{x}$. The Frechet derivative is defined as the linear operator on ω such that [Ortega70]

$$\lim_{\|\boldsymbol{\delta}\| \rightarrow \mathbf{0}} \frac{1}{\|\boldsymbol{\delta}\|} \|\mathbf{F}(\mathbf{x} + \boldsymbol{\delta}) - \mathbf{F}(\mathbf{x}) - \mathbf{J}_F(\mathbf{x})\boldsymbol{\delta}\| = 0. \tag{3.60}$$

With the pointwise description of $\mathbf{F}(\cdot)$ given in Eqn. (3.58) and the definition of the Frechet derivative given above, one can evaluate the derivative of $\mathbf{F}$, denoted as $\mathbf{J}_F$. A convenient norm on ω in which to evaluate $\mathbf{J}_F(\mathbf{x})$ from Eqns. (3.60) and (3.57) is

$$\|\mathbf{x}\| = \max_{\tau \in [0,t]} \left\{ \|\mathbf{x}(0)\|^{\Re^n} + \|\dot{\mathbf{x}}(t)\|^{\Re^n} \right\}, \tag{3.61}$$

where $\mathbf{x} \in \omega$, $\dot{\mathbf{x}}(t)$ is the usual derivative of $\mathbf{x}(t)$ with respect to t, and $\| \|^{\Re^n}$ is any norm on $\Re^n$.

Considering then a small perturbation in $\mathbf{x}$

$$(\mathbf{F}(\mathbf{x} + \boldsymbol{\delta}))(t) - (\mathbf{F}(\mathbf{x}))(t) = \frac{d}{dt}\boldsymbol{\delta}(t) - [\mathbf{f}(\mathbf{x}(t) + \boldsymbol{\delta}(t)) - \mathbf{f}(\mathbf{x}(t))]. \tag{3.62}$$

Expanding and approximating to order $(\|\boldsymbol{\delta}\|^2)$,

$$(\mathbf{F}(\mathbf{x} + \boldsymbol{\delta}))(t) - (\mathbf{F}(\mathbf{x}))(t) = \frac{d}{dt}\boldsymbol{\delta}(t) - \frac{\partial \mathbf{f}(\mathbf{x}(t))}{\partial \mathbf{x}}\boldsymbol{\delta}(t) + O(\|\boldsymbol{\delta}\|^2), \tag{3.63}$$

one can determine a closed form expression for the Frechet derivative to be

$$(\mathbf{J}_F(\mathbf{x})\boldsymbol{\delta})(t) = \frac{d}{dt}\boldsymbol{\delta}(t) - \frac{\partial \mathbf{f}(\mathbf{x}(t))}{\partial \mathbf{x}}\boldsymbol{\delta}(t). \tag{3.64}$$

Substituting the computed derivative from Eqn. (3.64) and $\mathbf{F}$ given by Eqn. (3.57) into Eqn. (3.59) and rearranging yields

$$\frac{d}{dt}\mathbf{x}^{k+1}(t) = \mathbf{f}\left(\mathbf{x}^k(t)\right) + \frac{\partial \mathbf{f}\left(\mathbf{x}^k(t)\right)}{\partial \mathbf{x}}\left(\mathbf{x}^{k+1}(t) - \mathbf{x}^k(t)\right) \qquad \mathbf{x}^{k+1}(0) = \mathbf{x}^k(0) = \mathbf{x}_0 \tag{3.65}$$

where $\mathbf{x}^k(t)$ is the k^{th} Newton iterate. We refer to Eqn .(3.65) as the Waveform-Newton (WN) algorithm for solving Eqn. (3.56).

The Waveform-Newton algorithm has a very simple interpretation: the differential equation is linearized about an initial guess waveform whose value at time zero matches the given initial condition for the differential equation. The guess waveform is then updated by solving the resulting linearized differential equation with an initial condition of zero. The original differential equation is then relinearized around the updated guess, and the process repeated until convergence is achieved. It can be proved that the rate of convergence of the Waveform-Newton algorithm is quadratic when the iterated guess is close to the correct value, which seems natural considering that Waveform-Newton algorithm is simply the function-space extension of the normal Newton-Raphson algorithm [Kantorovich64, White85]. For further reference about the algorithm, see [White85].

Discretizing Eqn. (3.65) with backward-Euler leads to a sequence of H linear equations, where H is the number of timesteps. The update equation at time t_{m+1}, where t_{m+1} is the timepoint corresponding to the m^{th} timestep, is given by

$$\begin{aligned} \frac{1}{t_{m+1}-t_m}\left(\mathbf{x}^{k+1}(t_{m+1}) - \mathbf{x}^{k+1}(t_m)\right) = \mathbf{f}\left(\mathbf{x}^{k}(t_m)\right) + \\ \frac{\partial \mathbf{f}(\mathbf{x}^k(t_m))}{\partial \mathbf{x}}\left(\mathbf{x}^{k+1}(t_{m+1}) - \mathbf{x}^{k}(t_m)\right). \end{aligned} \tag{3.66}$$

Rearranging Eqn. (3.66) leads to

$$\begin{aligned} \left[\frac{1}{t_{m+1}-t_m}\mathbf{I} - \frac{\partial \mathbf{f}(\mathbf{x}^k(t_{m+1}))}{\partial \mathbf{x}}\right]\left(\mathbf{x}^{k+1}(t_{m+1})\right) = \frac{1}{t_{m+1}-t_m}\mathbf{x}^{k+1}(t_m) + \\ \mathbf{f}\left(\mathbf{x}^{k}(t_m)\right) - \frac{\partial \mathbf{f}(\mathbf{x}^k(t_{m+1}))}{\partial \mathbf{x}}\mathbf{x}^{k}(t_{m+1}), \end{aligned} \tag{3.67}$$

where $\mathbf{I}$ is the identity matrix in $\Re^n$.

By examining Eqn. (3.67) it becomes clear that parallelism can be exploited evaluating H timesteps simultaneously. In fact direct observation of Eqn. (3.67) reveals that the evaluation and LU decomposition of the matrix on the left-hand side and the computation of all the terms on the right-hand side, except the $\frac{1}{t_{m+1}-t_m}\mathbf{x}^{k+1}(t_m)$ term, can be performed for all H timesteps simultaneously.

Moreover, the serial computation in the right is similar to that of a multiple timepoint algorithm. This serial computation involves determining $\mathbf{x}^{k+1}(t_1)$, which is then used to complete the right-hand side for the equation for $\mathbf{x}^{k+1}(t_2)$, from which $x^{k+1}(\tau_2)$ can be computed with a matrix back-solve, which is then used to complete the right-hand side of the equation for $x^{k+1}(\tau_3)$, etc. Rewriting the previous equation in a more compact form for all H timepoints

simultaneously, we get

$$\begin{bmatrix} \mathbf{A}_1 & 0 & \cdots & \cdots & \cdots & 0 \\ \mathbf{B}_2 & \mathbf{A}_2 & \cdots & \cdots & \cdots & 0 \\ 0 & \cdots & \cdots & \cdots & \cdots & 0 \\ 0 & \cdots & \mathbf{B}_m & \mathbf{A}_m & \cdots & 0 \\ 0 & \cdots & \cdots & \cdots & \cdots & 0 \\ 0 & \cdots & \cdots & \cdots & \mathbf{B}_H & \mathbf{A}_H \end{bmatrix} \begin{bmatrix} \mathbf{x}(t_1) \\ \mathbf{x}(t_2) \\ \vdots \\ \mathbf{x}(t_m) \\ \vdots \\ \mathbf{x}(t_H) \end{bmatrix} = \begin{bmatrix} \mathbf{C}_1 \\ \mathbf{C}_2 \\ \vdots \\ \mathbf{C}_m \\ \vdots \\ \mathbf{C}_H \end{bmatrix} \tag{3.68}$$

where

$$\mathbf{A}_m = \frac{1}{t_m - t_{m-1}}\mathbf{I} - \frac{\partial \mathbf{f}\left(\mathbf{x}^k(t_m)\right)}{\partial \mathbf{x}}$$

$$\mathbf{B}_m = \frac{1}{t_m - t_{m-1}}$$

$$\mathbf{C}_m = \mathbf{f}\left(\mathbf{x}^k(t_{m-1})\right) - \frac{\partial \mathbf{f}\left(\mathbf{x}^k(t_{m-1})\right)}{\partial \mathbf{x}}\mathbf{x}^k(t_{m-1})$$

Note that the the pattern of the matrix shown in Eqn. (3.68) depends on the integration formula used. However a block-band-lower-triangular matrix is always obtained. This is advantageous since not only the matrix can be calculated and stored in parallel, although it may need to be recalculated at each iteration, but also some method like Cyclic Reduction can be used to solve the system in $O(\log N)$ time.

3.3.2 Waveform Newton Circuit Simulation

Casting our methodology into the circuit simulation problem, we consider the system of differential equations in Eqn. (2.1) and follow the steps above. We are lead to a closed form expression for the Frechet derivative to be

$$\left(\mathbf{J}_F(\mathbf{v})\boldsymbol{\delta}\right)(t) = \frac{d}{dt}\left[\frac{\partial \mathbf{q}\left(\mathbf{v}(t), \mathbf{u}(t)\right)}{\partial \mathbf{v}}\boldsymbol{\delta}(t)\right] - \frac{\partial \mathbf{f}\left(\mathbf{v}(t), \mathbf{u}(t)\right)}{\partial \mathbf{v}}\boldsymbol{\delta}(t). \tag{3.69}$$

Substituting the computed derivative and $\mathbf{F}$ given by

$$\left(\mathbf{F}(\mathbf{x})\right)(t) = \frac{d}{dt}\mathbf{q}\left(\mathbf{v}(t), \mathbf{u}(t)\right) - \mathbf{f}\left(\mathbf{v}(t), \mathbf{u}(t)\right) \tag{3.70}$$

into the iteration equation for the Newton-Raphson algorithm for the solution of $\mathbf{F}(\mathbf{x}) = \mathbf{0}$ and rearranging yields

$$\frac{d}{dt}\left[\mathbf{q}\left(\mathbf{v}^k(t), \mathbf{u}(t)\right) + \frac{\partial \mathbf{q}\left(\mathbf{v}^k(t), \mathbf{u}(t)\right)}{\partial \mathbf{v}}\boldsymbol{\delta}^{k+1}(t)\right] = \mathbf{f}\left(\mathbf{v}^k(t), \mathbf{u}(t)\right) + \frac{\partial \mathbf{f}\left(\mathbf{v}^k(t), \mathbf{u}(t)\right)}{\partial \mathbf{v}}\boldsymbol{\delta}^{k+1}(t). \tag{3.71}$$

where again $\boldsymbol{\delta}^{k+1}(t) = \mathbf{v}^{k+1}(t) - \mathbf{v}^k(t)$.

The usual iterative stepsize refinement techniques used in conjunction with multiple timepoint methods may cause trapezoidal discretization to produce spurious oscillatory solutions in certain conditions. In order to prevent that, these algorithms are usually based on the more stable second-order backward-difference method defined as

$$y(t_{m+1}) = y(t_m + h_{m+1}) = \frac{(h_{m+1}+h_m)^2}{h_{m+1}(2h_{m+1}+h_m)} y(t_m) - \frac{h_{m+1}^2}{h_{m+1}(2h_{m+1}+h_m)} y(t_m - h_m) + \frac{h_{n+1}(h_{n+1}+h_n)}{2h_{n+1}+h_n} f\left(y(t_{m+1}), t_{m+1}\right) \tag{3.72}$$

where $h_m = t_m - t_{m-1}$.

Using this discretization the Newton-Raphson equation for H simultaneous timepoints yields

$$\begin{bmatrix} \mathbf{A}_1 & 0 & \cdots & \cdots & \cdots & \cdots & 0 \\ \mathbf{B}_2 & \mathbf{A}_2 & 0 & \cdots & \cdots & \cdots & 0 \\ \mathbf{C}_3 & \mathbf{B}_3 & \mathbf{A}_3 & 0 & \cdots & \cdots & 0 \\ 0 & \cdots & \cdots & \cdots & \cdots & \cdots & 0 \\ 0 & \mathbf{C}_m & \mathbf{B}_m & \mathbf{A}_m & 0 & \cdots & 0 \\ 0 & \cdots & \cdots & \cdots & \cdots & \cdots & 0 \\ 0 & 0 & 0 & 0 & \mathbf{C}_H & \mathbf{B}_H & \mathbf{A}_H \end{bmatrix} \begin{bmatrix} \boldsymbol{\delta}^{k+1}(t_1) \\ \boldsymbol{\delta}^{k+1}(t_2) \\ \vdots \\ \boldsymbol{\delta}^{k+1}(t_m) \\ \vdots \\ \boldsymbol{\delta}^{k+1}(t_H) \end{bmatrix} = \begin{bmatrix} \mathbf{D}_1 \\ \mathbf{D}_2 \\ \vdots \\ \mathbf{D}_m \\ \vdots \\ \mathbf{D}_H \end{bmatrix} \tag{3.73}$$

where $\mathbf{A}_m$ is the Jacobian of the circuit at timepoint t_m,

$$\mathbf{A}_m = \mathbf{J}_m^q + \mathbf{J}_m^c = \frac{\partial \mathbf{q}\left(\mathbf{v}^k(t_m), \mathbf{u}(t_m)\right)}{\partial \mathbf{v}} - \frac{h_{m+1}(h_{m+1}+h_m)}{2h_{m+1}+h_m} \frac{\partial \mathbf{f}\left(\mathbf{v}^k(t_m), \mathbf{u}(t_m)\right)}{\partial \mathbf{v}} \tag{3.74}$$

$\mathbf{B}_m$ and $\mathbf{C}_m$ establish the connection between the various timepoints,

$$\mathbf{B}_m = -\frac{(h_{m+1}+h_m)^2}{h_{m+1}(2h_{m+1}+h_m)} \mathbf{J}_{m-1}^q = \frac{\partial \mathbf{q}\left(\mathbf{v}^k(t_{m-1}), \mathbf{u}(t_{m-1})\right)}{\partial \mathbf{v}} \tag{3.75}$$

$$\mathbf{C}_m = \frac{h_{m+1}^2}{h_{m+1}(2h_{m+1}+h_m)} \mathbf{J}_{m-2}^q = \frac{\partial \mathbf{q}\left(\mathbf{v}^k(t_{m-2}), \mathbf{u}(t_{m-2})\right)}{\partial \mathbf{v}} \tag{3.76}$$

and $\mathbf{D}_m$ is the residue calculated as:

$$\mathbf{D}_m = \mathbf{R}_m^q + \mathbf{R}_m^c - \mathbf{R}_{m-1}^q + \mathbf{R}_{m-2}^q = -\mathbf{q}\left(\mathbf{v}^k(t_m), \mathbf{u}(t_m)\right) + \frac{h_{m+1}(h_{m+1}+h_m)}{2h_{m+1}+h_m} \mathbf{f}\left(\mathbf{v}^k(t_m), \mathbf{u}(t_m)\right) + \tag{3.77}$$

$$+ \frac{(h_{m+1}+h_m)^2}{h_{m+1}(2h_{m+1}+h_m)} \mathbf{q}\left(\mathbf{v}^k(t_{m-1}), \mathbf{u}(t_{m-1})\right) + \frac{h_{m+1}^2}{h_{m+1}(2h_{m+1}+h_m)} \mathbf{f}\left(\mathbf{v}^k(t_{m-1}), \mathbf{u}(t_{m-1})\right)$$

Results on the convergence of the Waveform-Newton algorithm are presented in the following theorem whose proof can be seen in [White85].

WAVEFORM NEWTON CIRCUIT SIMULATION

Theorem 3.4 *For any system of the form of Eqn. (2.1) in which* $\frac{\partial \mathbf{q}}{\partial \mathbf{v}}$ *is continuously differentiable, Lipschitz continuous, and has a uniformly bounded inverse with respect to* $\mathbf{v}$ *for all* $\mathbf{u}$*;* $\mathbf{f}$ *is continuously differentiable and Lipschitz continuous; and* $\mathbf{v}^0(t)$ *is a continuous, differentiable function such that* $\mathbf{v}^0(0) = \mathbf{v}_0$*; then the sequence of waveforms,* $\{\mathbf{v}^k\}$*, generated by the Waveform-Newton algorithm converges uniformly to the solution of (2.1).*

The iteration equations for the WN algorithm are time-varying linear differential equations and are therefore easier to solve numerically than the original system of nonlinear differential equations. For example, if the WN iteration equations are solved with the same discretization techniques as used for the classical direct method, then since the equations are linear, only a single matrix solution need be performed at each timepoint. Note however that the discretized WN algorithm is even more ill-suited to the simulation of large problems then the classical direct methods, because it requires the solution of an enlarged system of linear equations. In fact, for linear problems, assuming identical timesteps are used, the classical direct and the discretized WN methods are identical. However, the discretized WN can be much less efficient than classical direct methods when used to simulate circuits containing highly nonlinear elements. The difference is in how the two methods get the initial guesses for the Newton iteration. In the classical approach, generating an initial guess for the Newton method implies projecting the behavior of the system forward by one timestep. In WN, one projects the behavior of the system forward for an entire waveform. Therefore, in the classical method the initial guess is almost certainly in or near the region of quadratic convergence for the Newton method, whereas that would rarely be true for WN. A dramatic example of this problem can be found in [White85].

This seems to leave this methods at a certain disadvantage. However, from our perspective, the important point is that by using the Waveform-Newton algorithm, several timepoints are computed in parallel, which allows a more efficient use of a larger number of processors. Whether this will overcome the presented drawbacks is not clear.

There is another interesting context in which the Waveform-Newton can be seen. Recalling the Waveform Relaxation algorithm presented in section 2.4, we saw that it fully exploited multirate behavior by solving the differential equations in a decoupled fashion. However, each iteration of Waveform Relaxation involves solving nonlinear differential equations accurately, and is therefore computationally expensive. Although the point nonlinear relaxation methods presented in section 2.4 cannot fully exploit multirate behavior easily, they are in many cases more efficient than standard WR. This is partly because each relaxation iteration for those method is much cheaper, as the nonlinear relaxation iteration equations are solved approximately with a single Newton iteration. Applying a similar approach to WR results in the Waveform-Relaxation-Newton methods [White85] whose convergence properties can be shown to be similar to those of WR. The advantage here, again, is that the iteration equations for this

Waveform-Relaxation-Newton algorithm are time-varying linear differential equations which are easier to solve than the nonlinear differential iteration equations of WR.

3.3.3 Implementation Issues and Discussion of Results

In this section we will be concerned with the relevant design options focusing on the issues that also relate to a possible implementation of the algorithm on a massively parallel computer architecture. Our first concern in studying this type of algorithm for implementation on a massively parallel architecture was the fact that they, as we saw, increase the exploitable parallelism. However we must first address the question of whether the drawback of somewhat slower convergence in the presence of high nonlinearities can be overcome. We therefore decided to test the WN algorithm instead of its relaxation-based counterparts [White85]. Clearly this one is not suitable for the massively parallel architecture, but its study should provide some insight as to whether these methods are promising enough to pursue. In view of this decision, a direct method strategy was used to solve the enlarged linear system of equations resulting from applying the Waveform-Newton algorithm. If an implementation is attempted on the *Connection Machine*, some relaxation scheme will be used to exploit the parallelism in the problem.

The most important issues for this type of algorithms still remain unsolved. This included, as for any multiple timepoint algorithm, the question of timepoint positioning and the convergence checking criteria. In fact, besides the method used to approximate the solution of the system of equations in Eqn. (3.73), the timestep control strategy may also greatly influence the results to be obtained. Typically, for Waveform-Newton iterative solution, the first few iterations are performed on a coarse-grid that is later refined as the iterations progress, in accordance with the calculated local-truncation error. This adaptive timestep strategy can be very efficient since the use of the *"a posteriori"* error estimates available from previous iterations can lead to a better placement of the timepoints [White85]. However our main concern is to efficiently exploit all the resources and therefore we need to keep all the processors busy. This seems to indicate that we would be interested in setting up an *"a-priori"* fixed number of timepoints whose time positioning could possibly change during convergence and error checking. In addition, to avoid excess communication, the adaptive timestep strategy should ensure that the layout of data onto the processors is minimally disturbed.

Clearly, the adopted solution algorithm will influence the layout of the data onto the processors. The two most attractive approaches, namely coarse-grain matrix block to processor allocation versus fine-grain device and node to processor allocation, should be considered. The first approach wastes some of the available parallelism, which is not necessarily a bad choice if the problem is large enough. On the other hand, the second approach requires more communication steps which, when performed randomly, is a very expensive operation on the a massively parallel computer.

As a first approach to the problem, we tried to work around the timepoint positioning

problem together with the error and convergence criteria methods. We therefore started by implementing the Waveform-Newton algorithm merely by causing a multiplication of the circuit data (which although wasteful of memory, makes sense on a massively parallel computer architecture) and getting the enlarged system shown on equations (3.73). This system was solved with direct methods on a sequential implementation, since that is irrelevant for the issues we are considering.

For our studies we have used a set of test circuits whose characteristics can be seen in Table 3-1. For those circuits, Table 3-2 summarizes some of the results obtained using 2^{nd} order backward differentiation formula (BDF).

Circuit	# of nodes	# of internal nodes	# devices
ckt	23	21	43
cramb	149	58	1203(933)
cramc	703	612	1782(1678)
decpla	66	56	510(442)
domino	12	6	77(48)
dtest	6	5	41(32)
modeol	601	600	1200
newop	52	40	226(196)
opamp	13	8	70(52)
parop	192	180	374
pres	11	9	20
rctest	11	8	20
scdac	155	150	1223(958)
shmem	759	740	6192(3823)
webop	52	40	226(196)

Table 3-1: Circuit Data for the test circuits.

Our first approach for solving the problem was to naively consider the enlarged system as if it were the result of formulating the equations of a single circuit. We used equally spaced timepoints within the simulation window and decided that checking for convergence would be equivalent to enforcing simultaneous convergence on all timepoints, whereas determining the error would be equivalent to determining the maximum error on all timepoints. This approach, although simple, is clearly not very satisfying as the results of the next tables show. Table 3-3 presents the results obtained using the 2^{nd} order backwards differentiation formula.

In fact the strategy proposed has two major drawbacks. For one, by enforcing simultaneous convergence in all timepoints we will not accept partial convergence, i.e., even if the very first few timepoints have converged, we will say that the whole batch has not converged simply because the latter ones have not. This is clearly bad in the case where we will have some major

Circuit	# Newton iters	# of Timesteps	Time
ckt	184	117	2.73333
cramb	3362	944	402.4
cramc	4090	1124	1113.52
decpla	361	126	22.1
domino	840	245	8.48333
dtest	840	245	6.15
modeol	90	63	13.2667
newop	412	176	15.2833
pamp	460	308	3.83333
parop	544	195	42.2
pres	103	81	1.55
rctest	60	51	0.783333
scdac	3628	1021	382.983
shmem	641	198	448.9
webop	414	176	11.6333

Table 3-2: Simulation data with the 2^{nd} order BDF.

Circuit	# Newton iters	# of Timesteps	Time
ckt	378	830	24.95
cramb	6919	12450	7833.3
cramc	8606	13360	45047.4
decpla	699	1480	286.883
domino	1667	1970	58.7667
dtest	1667	1970	45.8333
modeol	84	190	1303.18
newop	577	1210	208.2
opamp	8897	32895	390.35
parop	635	1205	1067.57
pres	170	455	8.23333
rctest	50	85	1.56667
scdac	7139	11430	7943.63
shmem	1563	2875	21487.8
webop	568	1195	195.517

Table 3-3: Simulation data with waveform-newton using the 2^{nd} order BDF.

change in the circuit behavior in the middle of our interval, a situation that is bound to occur in general. Also, by determining the local truncation error as the maximum error throughout the interval instead of determining it locally, we will cause the timesteps to shrink even at timepoints for which the local error was acceptable. These two problems can be seen on the data shown on the tables by noting that the number of timesteps used for the simulation of the Waveform-Newton version is much larger than the number of those used for the pointwise version. The fact that the number of Newton iterates is also larger is therefore a consequence of using more (smaller) timesteps to simulate the same time interval.

In order to solve these problem we implemented a new version that addresses those questions. On one hand we check convergence locally at each timepoint and accept without changes any converged timepoint even if the following ones have not reached convergence. Also when determining the error, we also do a local verification and change the timesteps only for those timepoints whose error exceeds the maximum allowed truncation error.

New questions arise when implementing this strategy. The first one is related to the ratio of the Newton iterations per checking, i.e., how many Newton iterations should be performed on the enlarged system before calculating the error. Also, an important point is whether the most significant criteria is the error estimation or the convergence criteria. In our first experiments, we decided we would check the local truncation error after each Newton iteration. This approach has the advantage that we do not waste time iterating on the Newton loop whenever the answer seems to be very far from the predicted behavior.

In fact is was seen that early Newton iterates can be far from the correct solution, and the extra effort in trying to get closer to the solution is not compensated. On the other hand, for some circuits presenting rather rapid and sudden node changes, we will most certainly reject large timesteps when in fact, if a few more Newton iteration were performed, convergence could start to appear. Nevertheless we observed, that if by default a minimum number of Newton iterations (greater than one) is enforced before checking the error, considerable effort may be wasted for those cases where the Newton iterates are far from the correct solution.

From our preliminary results we have been able to see these two effects showing in specific examples. Specifically, by enforcing the local truncation error (LTE) checking after each Newton we can quickly reject large timesteps in regions of rapid changes, but are still lead to convergence difficulty in other problems. If on the other hand we enforce that a given number of Newton iterations (say 2 or 3) must be performed before LTE checking we are able to achieve convergence on those examples but get poorer performance on the other ones, due simply to the larger number of total iterations.

The question of LTE checking is also not without its details. The point being that we have assumed that the number of timepoints is always fixed, it seems that if a later timepoint has an acceptable error, but a previous one does not, we will have to neglect the results obtained for the acceptable one since all the timepoints will move back to the region where problems

are being detected. For this reason we do not even check the error after we have found a rejectable timestep. Clearly on a parallel implementation all the timepoints will be checked simultaneously and this question becomes meaningless, but nevertheless the same phenomenon of rejecting possibly acceptable timepoints because the previous one were too much in error will occur.

In general the algorithm being used goes as follows: after the necessary number of Newton iterations the local truncation error at each timepoint is determined; if the error is below the maximum acceptable, the timestep is accepted; otherwise, if it exceeds that predetermined value, the timestep is rejected and the local truncation error is used to shrink it. It would seem natural to increase the timestep whenever the LTE is acceptable. However, this would cause problems since when enlarging the timestep we would have to make sure we do not go past the next timepoint which may even be rejected. So, it seems conservative, but nevertheless wise, to just keep the timestep whenever the error is acceptable.

Another strategy can be implemented to take advantage of this early timepoint convergence. If after a few iterations the first few timesteps have converged, one could consider sliding forward the time-window containing the timepoints. This however implies some rearrangement of data within the processors, which in turn implies some communication, and therefore may not necessarily be compensating.

A problem that was detected in the early stages of the implementation was the fact that our setting of the timepoints did not necessarily comply with the input breakpoints. In fact since we are simulating simultaneously several timepoints, there is no explicit guarantee that the timepoints will land exactly on the input breakpoints. A version that guarantees catching all breakpoints at the expenses of using smaller windows in some points was developed but the results must be considered worse than previously basically because some parallelism is lost when using smaller window sizes. Therefore it seems more efficient not to enforce the catching of all breakpoints, a highly undesirable situation.

The issues presented and discussed above are but some of the most important issues relating to this work. At this point it becomes extremely important to determine the amount of speedup it is possibly to achieve when implementing a linear system solver on a massively parallel computer architecture. If this will lead to a significant improvement over the sequential version, then the above described issues, such as input breakpoint enforcing, window sliding, Newton iterations per error checking, will all have to be studied in more depth in order to get the greatest performance out of our algorithm.

Based on the results present and the objections stated, it did not seem at this point that using multiple timepoint methods on a massively parallel computer architecture would be a good choice. We state this belief mainly based on three points:

- The communication needs for these methods are larger, which can easily be seen by the fact that the matrix is larger;

- There seems to be no simple and effective way to place the timepoints "a priori" which has been verified to cause too small timesteps to be taken;
- The convergence properties of these methods are weak, in the sense that for some circuits it is in fact better to follow a pointwise approach.

We therefore decided to end our study of these methods. Furthermore, since previous experience seem to indicate that the price to pay in terms of communication, for having a nonstructured and irregular problem is excessive we decided to focus on more regularly patterned problem.

4

The Connection Machine

The *Connection Machine*® system[1], is a commercially available massively parallel supercomputer developed by *Thinking Machines Corporation* [Hillis85]. The *Connection Machine* is a data parallel computer which consists of an instruction engine and multiple data processors. The data processors are capable of sharing data by using a high-speed communications network while the instruction engine is responsible for broadcasting instructions and addresses to each of the data processors and for executing the control aspects of the program. Users interact with the instruction engine through a standard front-end serial computer.

4.1 Architectural Overview

Employing anywhere up to 65,536 individual 1-bit processing elements, the second generation *Connection Machine* system, the *CM–2* has a reported performance above 2500 million instructions per second (MIPS). The total memory capacity of a fully integrated system is 512 megabytes where each processing element has access to 32k bytes of local memory [CM2]. Figure 4-1 shows the detail of a *CM* processor.

The bit-processors and their local memory are arranged in hardware with 16 to a chip. They are interconnected by a high bandwidth network able to perform data transfer rates to storage of approximately 45 Gbytes/s at a clock rate of 7 MHz. The *CM–2* can be equipped with 2048 Weitek floating-point processors, where every two of the bit-processor chips share a floating point unit, i.e. every 32 bit-processors share a single Weitek unit. With appropriate vectorization, this indicates a potential peak-performance of around 2500 MFLOPS (Millions of Floating Point Operations Per Second).

The *Connection Machine* is a "single-instruction multiple-data" (SIMD) parallel computer [Flynn66]. The operations of the individual processors are controlled by a sequencer that converts the commands given by the front-end. The front-end executes the control structure of

[1] *Connection Machine* is a registered trademark of *Thinking Machines Corporation*

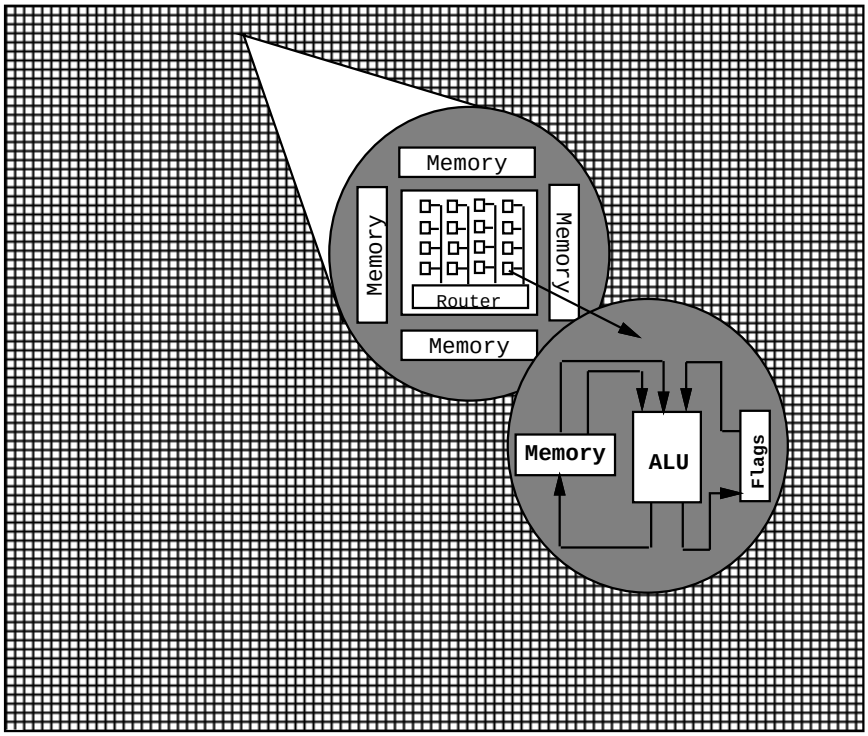

FIGURE 4-1: CM data processor. (Copyright *Thinking Machines Corporation*. Used with permission)

programs, issuing commands to the *CM–2* processors whenever necessary. For faster execution the front-end interfaces to a ***sequencer*** generating a stream of macro-instructions that is translated into microcode and broadcast to the processors for execution. Figure 4-2 presents the computational model of the *Connection Machine*.

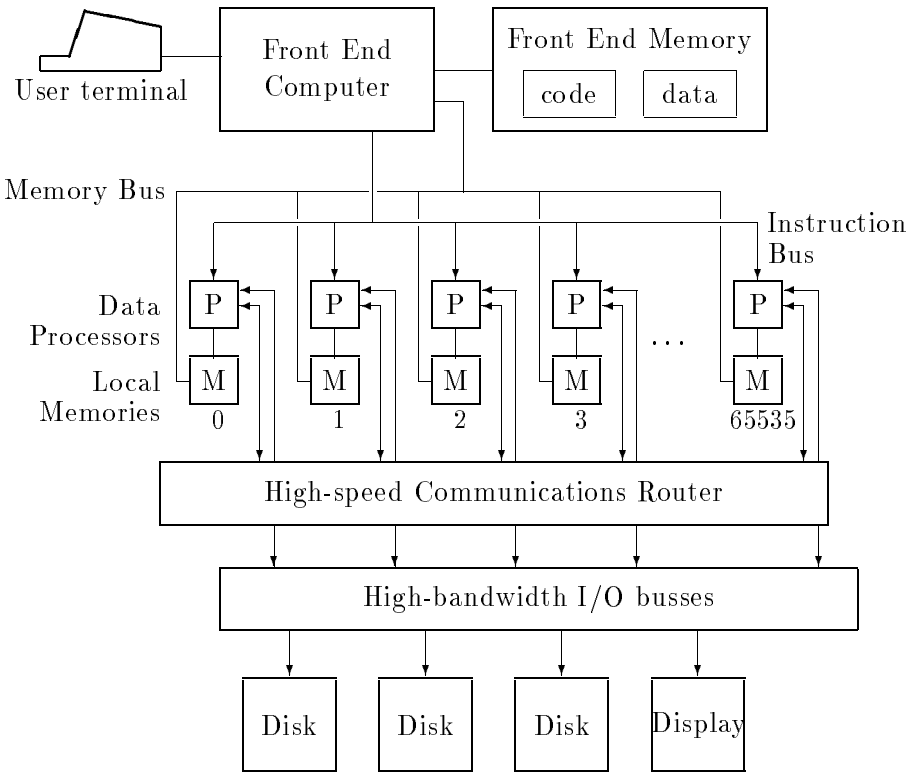

Figure 4-2: The Connection Machine Computational Model.

The 64k processors are connected in a 12-dimensional Boolean cube network, and there is hardware to support general parallel communication of data between processors. The high level languages provide two mechanisms by which the processing elements of the *Connection Machine* system can communicate information among themselves: lattice emulation and general routing. For communication in arbitrary patterns, the machine is equipped with a router which selects one of the shortest paths between source and destination; contention for communication channels is resolved in several ways. The lattice emulation makes use of the fact that lattices with powers of two axes lengths are subgraphs of Boolean cubes. As an example of lattice emulation, ***grid*** structures permit the processors to be arranged in an array of arbitrary size (constrained to be a power of two) and dimensionality. Each processor can communicate with its nearest neighbors in this multidimensional grid. The hypercube connection provides communications for algorithms with an irregular message structure, while the grid is much more efficient for algorithms with more structured communication patterns. Clearly, regular

communication, along the cube wires, is much faster than irregular communication using the router due to possible collisions along the routed path and to the router overhead.

4.2 Virtual Processors

Array elements in the *Connection Machine* programming languages are often referred to as *virtual processors* [Hillis85, CM2]. In general, several virtual processors (array elements) are mapped to the storage of each physical processor. The number of virtual processors per physical processor is called the *virtual processor ratio*. The storage of a physical processor is divided among the many virtual processors. The system software for the *Connection Machine* allows transparent use of virtual processors. The abstraction of virtual processors is supported by segmenting the local memory of each processor, and by having the sequencer multiplex over these memory segments as necessary. This is transparent to the programmer of the front-end who always deals with virtual processors. This has the advantage that the same program can be run unchanged on different sizes of *Connection Machine* systems. Of course there will be a degradation in speed that is, in the worst case, linear in the number of virtual processors used; this is because the different virtual processors within a single physical processor operate sequentially. In the virtual processor mode, the application program is assigned as many virtual processors as there are data objects, at the cost of reduced memory per virtual processor. This feature alleviates the applications program from the responsibility of conforming to the number of physical processors. Figure 4-3 exemplifies the use of virtual processing on the *Connection Machine*.

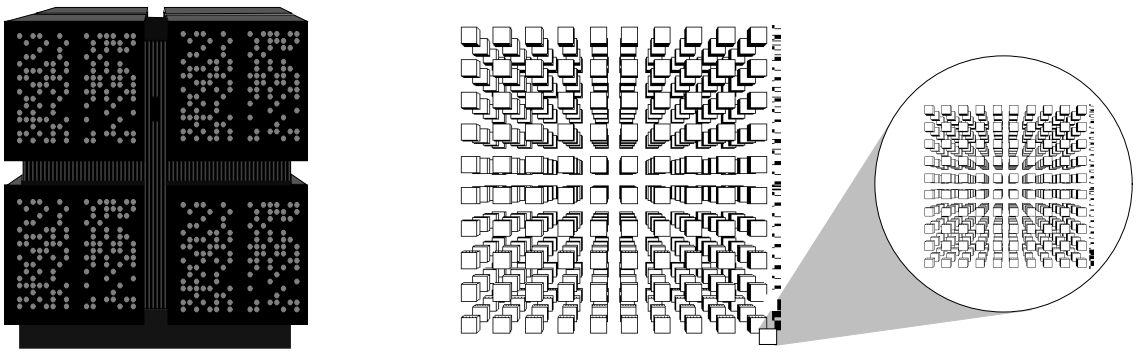

FIGURE 4-3: Virtual processing on the *Connection Machine*. (Copyright *Thinking Machines Corporation*. Used with permission)

4.3 Data Parallelism

Of the different dichotomies that may be used to classify parallel architectures, Flynn's [Flynn66] characterization makes a clear distinction between SIMD machines (single instruction,

multiple data) and MIMD machines (multiple instruction, multiple data). Another useful dichotomy is that of fine-grained parallelism versus coarse-grained parallelism. Yet another is that of data-level parallelism versus control-level parallelism.

To some extent, these dichotomies are isomorphic to one another. In fact, SIMD machines tend to be fine-grained, data-parallel devices. Problems that deal with large amounts of data, for which the application of data-level parallelism is best suited, also require a large number of processors and hence a fine-grained parallel processor. Synchronization and logistical problems with broadcasting an instruction stream to each of these many processors explains the tendency for such fine-grained parallel processors to be SIMD. On the other hand, MIMD machines tend to be coarse-grained, control-parallel devices for the obvious converse reasons.

Data parallel computers like the *Connection Machine* system achieve increased performance by taking advantage of parallelism found in the data of a problem. Data parallelism is a conceptual approach to programming a parallel machine which emphasizes the parallelism inherent in the problem when the amount of data over which operations are to be performed becomes large. Rather than the length of code, the focus in data-parallelism is in the amount of data. The most commonly referred example is that of image processing whereby a single function requiring many sequential steps must be applied to all the pixels of an image and for which massively parallel processing can produce a speedup close to the number of image pixels, independent of the length of the serial code that each pixel must pass through. This example illustrates that in fact data-parallelism does not abridge Amdahl's law and should not be viewed in that context.

The data-parallel programming model of a massively parallel computer can be viewed as the counterpart of the usual assumption made when programming sequential machines, regarding the availability of memory. Data-Parallelism, together with the existence of a large number of processors, or of virtual processors, allows the programmer to make the assumption that the number of available raw processors is infinite and that there will always be "enough" processors for whatever task is at hand.

The concept of data-parallelism is therefore intrinsically associated to that of massively parallel processing. In a data-parallel implementation, one typically assigns a processor to each data element of the problem and then selectively executes a sequence of code on a subset of the available processors. The possibility of selecting processors to execute certain pieces of code requires that others are made inactive during the computation, due to the SIMD nature of these machines. On the *Connection Machine* this can be accomplished in several ways but always by means of setting or resetting a context flag within each processor. All processors which are selected actually execute the code, while those that are not selected do nothing. For example, to implement an "if" statement, first, all processors evaluate the condition. Second, only those processors for which the condition evaluated to true are selected and the "then" clause of the program is executed. Next, only those processors for which the condition evaluated to false are selected and the "else" clause is executed. Note that when the program execution has

multiple clauses, all of the clauses must be executed one after the other to cover all possible configurations of data on the processors.

Data-Parallelism requires the notion of local storage within a single processor and *connectivity* among the processors. Communication in the *Connection Machine* is as described in section 4.1 provides for local and global references. Local storage was also described. Of particular interest is the concept of a parallel variable, i.e., a variable which has an instance in every processor. When operations are performed on parallel variables on the *CM*, all the operations are performed simultaneously, one in each processor. Figure 4-4 shows an example of using data-parallelism to solve a problem with a large number of data elements.

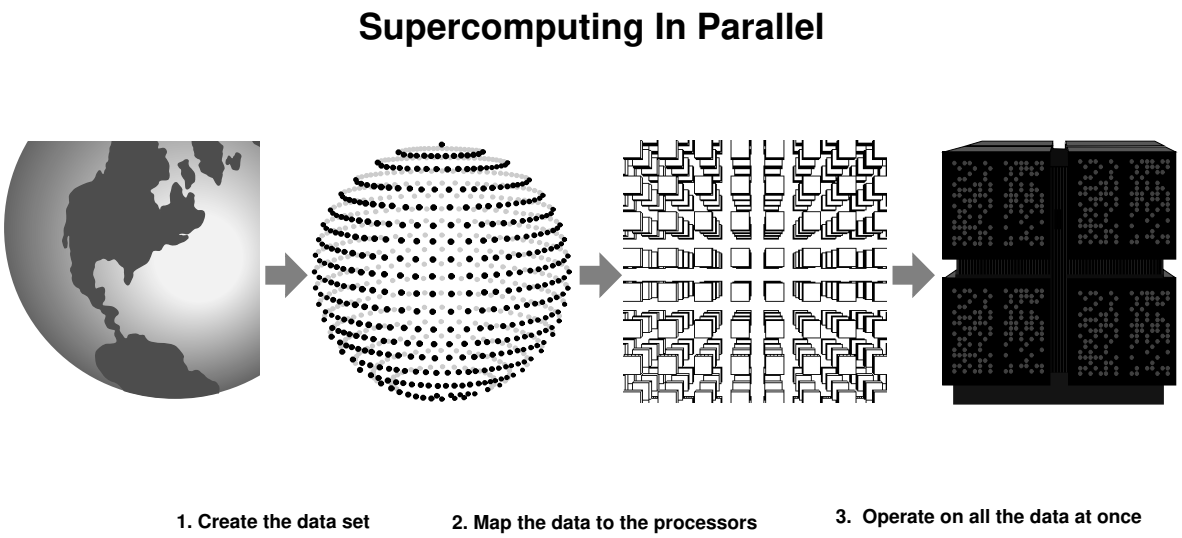

FIGURE 4-4: Data-parallelism on the *Connection Machine*. (Copyright *Thinking Machines Corporation*¡. Used with permission)

4.4 Programming the Connection Machine

To be used most effectively, the *Connection Machine* should exploit parallelism at the level of the data on which it operates; that is, it is very efficient for tasks that can be solved by simultaneous operation on all of the data elements involved. These types of problems are easily decomposed on a very fine grain whereby each data element can be characterized as belonging to one of a few data classes. With generality, we can state that the class of problems which can be solved efficiently on the *Connection Machine* are those which exhibit a high degree of *data parallelism*, i.e.,

- One can map individual pieces of data to individual processors identically for all relevant processors
- One can operate identically on the data with all the relevant processors

Depending on the structure of the problem data and the computations involved, the pattern of the layout of the data elements to the processors may vary. This layout must take into consideration the communication needs of the underlying algorithm to be executed. Achieving the greatest locality in computations will lead to more efficient implementation, while sharing values between processor elements will incur some performance penalty.

There are several levels of abstraction for communication between processors on the *Connection Machine*. Grid communication, also known as NEWS communication (for North, East, West and South) allow neighbors on a grid to exchange values, and, and by repeated calls, processors any distance away can also exchange data. However each processor can only receive data from the same relative location as all of the other processors that are currently requesting a message. This limitation can be overcome by using the hypercube message passing system, for which any set of processors can send messages to any other set of processors simultaneously, with the hardware controlling the actual route taken by the messages and handling of collisions. Using the general router system allows any processor to send or receive an arbitrary message from any other processor. This capability makes the *CM* more general than other previously designed massively parallel processors, because it allows many messages to be sent to many different relative locations simultaneously and automatically, albeit at the cost of some performance loss. Nevertheless, this allows the inexact embedding of arbitrary data configurations onto the hypercube, with little effort from the programmer, a feature that has been the basis of several current implementation of many algorithms [Webber87].

Since most implementations make use of the general routing facilities, it is educational to make some comparative studies regarding the performance of some simple operations on the *CM* . The values presented in Table 4-1 refer to 800 repetitions of a simple matrix-vector product on a $(64)^2 \times (64)^2$ matrix, where each unknown represents a point in a 64×64 grid. Structurally, this matrix is tridiagonal matrix plus a lower and upper diagonal at a distance 64 from the diagonal. This matrix can be efficiently laid out on the *CM* as discussed in greater detail in chapter 5. For our discussion here, it is enough to say that each processor on a two-dimensional grid handles one row of the matrix and that a matrix-vector product amounts to eight local computations together with four communication steps. Our goal with these measurements is to obtain empirical values for the communication costs versus the cost of performing local floating point operations. The *CM* results presented were obtained respectively with router communication and NEWS communication. A last set of results was obtained whereby the communication steps were omitted. This allows us to determine ratios of communication versus computations for this type of tasks.

The above results should be viewed only as a measure of the relative costs of computation and communication. Comparing the router times to the NEWS (nearest neighbor in a two-dimensional grid) times, we can see that the cost of communicating with the router exceeds by almost one order of magnitude the cost of performing nearest neighbor communication via the

Experiment	CPU Time (seconds)
Front-End	53.02
CM (router communication)	5.58
CM (NEWS communication)	0.92
CM (no communication)	0.28

Table 4-1: Using Matrix-Vector products to measure primitive operations.

NEWS grid. Furthermore, if the comparison is taken for non-nearest neighbor communications the ratio will increase leading to further performance degradation. From the results shown, we can also infer the cost of NEWS communication versus floating point operations. As stated, a matrix-vector operation with the data to processor layout described above requires 2 communication steps to get the neighboring values, plus eight floating point operations and finally two more communication steps to send the values back. This gives a ratio of 1:2 in communication versus computation. From the last measurement we can see that the computation cost is about one third of the overall cost which implies that the cost of a NEWS communications is four times the cost of a floating point operation (note that these results are for single precision arithmetic, but they will scale similarly for double precision arithmetic).

The results presented in Table 4-2 were obtained from an experiment aimed at determining the obtainable floating-point performance of the machine. The experiments were coded in *C* *Paris* (*Paris*, which stands for PARallel Instruction Set, is the machine's top level assembly language). The front-end experiments were run on a conventional Sun-4 workstation and the *CM* results were obtained on a 16K *CM–2* with double-precision floating point hardware. All computations were performed in double precision arithmetic. This experiments allowed us to measure the expectable raw performance, determine how the computation cost of both communication and floating point operations scale with the size of the problem at hand, and also gave a rough idea of the machine speedup that could be obtained versus a conventional computer. The results shown are the elapsed times required to perform 100 repetitions of the matrix-vector product described above. The columns shown represent the sequential computation time and the *CM* time. A third column shows the *Connection Machine* time of the same experiment but while omitting the communications steps.

Comparison of the first and second columns allows us to extrapolate the speedups that are possible if efficient implementations are performed on the *Connection Machine*. The third column gives us a measure of the raw floating-point performance attainable, while comparison of the second and third column reveals how communication time and floating-point computations scale with increasing problem size and also with different VP ratios.

Problem Size	CPU Time (seconds)		
	Front-End	CM (vp ratio)	CM[1] (vp ratio)
64 × 64	12.72	0.13 (1)	0.05 (1)
128 × 128	53.52	0.20 (1)	0.05 (1)
256 × 256	216.95	0.48 (4)	0.13 (4)
512 × 512	867.8[2]	0.78 (16)	0.49 (16)
1024 × 1024	3471.2[2]	3.24 (64)	1.95 (64)

Table 4-2: Measuring speedup, raw performance and scalability with Virtual Processor ratio.

Noting that each iteration of the matrix-vector product involves 8 floating-point operations, the results in the third column point to a performance of roughly 430 MFLOPS on the 1024 × 1024 problem. Extrapolation to a full 64K machine indicates a peak performance around 1.72 GFLOPS. This results agree with the experiments in [Levit88] which indicate that although the peak performance of the machine is around 25 GFLOPS, obtainable performance from Paris is below somewhere 2.5 GFLOPS.

Comparison of the results in the second and third columns allows us to verify, at least for low VP ratios, our previous claim that the cost of a communication step exceeds by a factor of four the cost of floating-point computation.

The sequential results seem to indicate that as the size of the problem duplicates the computational cost increases fourfold. The extrapolated results, when compared with the *CM* results show that for a matrix of size 512 × 512 a speedup of over 1200 is possible. Moreover the extrapolated sequential results are optimistic since contention for memory resources will make the sequential version even slower.

From the above results we can infer that increasing the size of the problem is not accompanied by a proportional increase in computation time. In fact, the increase in problem size only leads to increasing time due to the penalty factor paid for using virtual processing. From the results above we can see that doubling the virtual processor ratio causes a sublinear increase in computation time, more noticeably for small VP ratios (a factor of 1.5 is observed both in communication time and floating-point performance). For larger VP ratios communication becomes a less important factor since many equations will be on a given processor.

The above results corroborate those obtained in other reported performance analysis of the *Connection Machine* [Mcbryan88, Levit88, Myers88] and lead us to believe that a problem is better suited for the *CM* if, in addition to exhibiting data-parallelism, it is also local and regular. By this we mean that communication between processors should take place in a regular

[1] No communication performed.
[2] Extrapolated results based on the results obtained for smaller size matrices.

and localized fashion (e.g., a 2-dimensional grid, nearest neighbors on a cube, etc.). These considerations have guided our work and have been overarching constraints on our decision making.

5

Massively Parallel Circuit Simulation

5.1 Introduction

This chapter covers the implementation and results obtained from our experience implementing circuit simulation algorithms on the *Connection Machine*. Section 5.2 describes our first approach to an implementation on the *Connection Machine* and parallel linear systems solvers are compared for efficiency. Next, in section 5.3 we present a set of visions circuits, describe the tasks they accomplish and the features that make them distinct. An implementation on the *Connection Machine* is finally presented, that is able to simulate these type of circuits within acceptable time.

5.2 Generic Circuit Simulation on the Connection Machine

Circuit simulation is a very computationally expensive problem which presents some features that are desirable for implementation on the *Connection Machine*. Simulating current designs requires large amounts of data, but the simulation problem has an underlying fine-grain structure and only a small number of different types, namely nodes and devices. At first approximation one can devise an implementation in which each data type is mapped to a unique processor, and the interconnect between the nodes and devices is represented by pointers to other processors. Superposition of types is also possible depending only upon processor memory and data-to-processor layout.

The primary computational tasks for a direct method solution consists of device evaluations and the solution of a sparse system of linear equations. The computation involved in device evaluation maps very well onto this parallel processing structure, in that by selecting the processors that hold devices of a given type and executing the code for evaluating that

type, *all* devices of that type are evaluated simultaneously. Synchronization or load balancing are basically non-existent concerns, since all computations occur at exactly the same time, and everything is completed with just one function call. The fact that there are more than one type of devices is in essence a deviation from the *SIMD* paradigm, since not all processors will hold devices of the same type and therefore some evaluation loop must be set based upon the various device types and their various regions of operation. However this problem has always been neglected since the number of device types is usually very small compared to the number of devices. Performance-wise this means that a constant factor is lost, unless some overlapping exists, which may become a significant factor if the number of device types and regions of operation is large.

Solving the sparse matrix, however, is not an easy task on the *Connection Machine*, because of the complexity of the algorithm and the amount of communication required for the computation. If the matrix were dense, the implementation would be easier because the data and the communication pattern would be more regular and the required computation would be known beforehand. However in the sparse matrix case, the computation required is dependent on the structure of the matrix. Solving the sparse matrix is, in fact, the bottleneck of any implementation and it will undoubtedly constrain the data to processor layout. Since the computation required for sparse gaussian-elimination matrix solution is relatively expensive compared to the relaxation methods described, many relaxation iterations can be done during the time required for one matrix solution. Therefore relaxation methods seem to be better suited as solution methods for these architectures.

Massively parallel circuit simulation has already been attempted with mixed results [Webber87]. The approach pursued by Webber was based on the point Gauss-Jacobi nonlinear relaxation method. Overall, however, his approach was not an optimal one for the *Connection Machine*. In his implementation, the circuit is partitioned at a very fine grain with circuit nodes and the various types of devices being the data classes operated upon. All devices in each class are linearized and and evaluated in parallel, which by itself leads to a great acceleration since only a few classes exist. The solution method adopted was also optimal in the sense that it took advantage of the resources available.

Since nonlinear relaxation was used, only one relaxation iteration is performed per Newton iteration. This is an important point, since by using this approach only the diagonal entries of the matrix and the right hand side have to be calculated. The matrix solution step is then only a parallel scalar divide operation. After each relaxation iteration the newly computed guess is propagated using the *Connection Machine* scan operators and a new iteration can be started.

A consequence of using the nonlinear Gauss-Jacobi relaxation method is that only the diagonal entries of the matrix need to be calculated or stored, so there are fewer matrix entries to calculate during the device evaluations and no processors are required for storing the off-diagonal entries. Also a simple matrix solution technique can be used, Although the the methodology

has the advantages mentioned, the algorithm has some serious drawbacks. In fact convergence can be very slow which may have a dramatic impact on the attainable performance. Also, Gauss-Jacobi relaxation can be unreliable because in some cases convergence may be indicated when the result is far from the correct solution.

The results reported in Webber's work show that, in general terms, the penalty to pay for using a point decomposition method offset the savings got from the parallel computations. This seems to imply that obtaining the most parallelism out of a problem by using a methodology with slow convergence may not be a successful combination, and therefore that some block algorithm should be considered.

5.2.1 Nonlinear Band Relaxation

A more recent approach was proposed as an alternative to Gauss-Jacobi relaxation [Lumsdaine88]. This algorithm, called "Band-Relaxation" was aimed at improving the reliability of Gauss-Jacobi relaxation by extracting a variable-sized band from the matrix and solving that band directly. This leads to a parallel relaxation algorithm which for special cases has provably better convergence properties than Gauss-Jacobi. Furthermore, this algorithm can be used effectively on a massively parallel computer, because band matrices can be solved in $O(\log n)$ time given $\frac{n}{2}$ processors, where n is the number of equations in the system [Dongarra87, Duff86].

To describe the algorithm we now make a simple detour and follow a slightly different path. We have seen that when using direct methods, the iterative Newton method is used to solve the nonlinear algebraic problem associated with each timestep of the transient analysis of a circuit, and each that iteration of the Newton method involves computing the solution to a sparse linear system. Specifically, for each Newton iteration an $\mathbf{x}$ must be found such that $\mathbf{Ax} = \mathbf{b}$, where $\mathbf{A} \in \Re^{n \times n}$ is the Jacobian of the nonlinear system, $\mathbf{x} \in \Re^n$ is the vector of node voltage updates, and $\mathbf{b} \in \Re^n$ is the Newton residue.

The Gauss-Jacobi relaxation applied to the solution of this this linear system can be written compactly as

$$x_i^{k+1} = \frac{1}{a_{ii}} \left[b_i \; - \sum_{j=1, j \neq i}^{n} a_{ij} x_j^k \right], \qquad i \in \{1, ..., n\}, \tag{5.1}$$

where k is the iteration index and a_{ij} is the ij^{th} entry of the matrix $\mathbf{A}$.

As the equations represented by the rows in $\mathbf{A}$ are solved independently, Gauss-Jacobi can again easily exploit n processors effectively. However, there are two difficulties with linear Gauss-Jacobi relaxation: it is inefficient as the timestep required to achieve convergence may be very small; and it is unreliable because false convergence is common due to the limited spread of information with each iteration [Deutsch85, Saleh87, Webber87, White86].

To see how to construct more reliable relaxation methods with nearly as much parallelism, it is helpful to cast relaxation into a more general frame with the following definition.

SPLITTING OF A MATRIX

Definition 5.1 *Any two matrices* $\mathbf{M}, \mathbf{N} \in \Re^{n \times n}$ *are said to be a splitting of the matrix* $\mathbf{A}$ *if* $\mathbf{A} = \mathbf{M} - \mathbf{N}$.

The iteration equation for a relaxation algorithm based on a given splitting is then

$$\mathbf{M}\,\mathbf{x}^{k+1} = \mathbf{N}\,\mathbf{x}^{k} + \mathbf{b}, \tag{5.2}$$

and the asymptotic rate of convergence is the spectral radius of $\mathbf{M}^{-1}\mathbf{N}$, $\rho(\mathbf{M}^{-1}\mathbf{N})$ [Varga62].

It becomes clear that if the splitting is chosen such that $\mathbf{M}$ is easy to factor and the convergence of Eqn. (5.2) is good, we obtain an efficient method for solving the system of equations. Note that $\mathbf{M}$ should be easy to factor in a parallel fashion, since the objective is to produce an efficient parallel algorithm.

For the case of Gauss-Jacobi, $\mathbf{M}$ is a diagonal matrix whose elements are the diagonals of $\mathbf{A}$ and is already in factored form. This leads to a very efficient parallel algorithm, but one which has poor convergence properties. One would expect that movement of off-diagonal elements of $\mathbf{A}$ from $\mathbf{N}$ to $\mathbf{M}$ would accelerate the convergence, because more of the system is being solved directly. This conjecture can be proven for the case of an $\mathbf{A}$ which is diagonally dominant with positive diagonals and negative off-diagonals [Varga62].

If a band relaxation scheme is used, the ordering of the equations in $\mathbf{A}$ will determine which off-diagonal elements are in the band, so it is possible to change the convergence properties of the relaxation by reordering $\mathbf{A}$. Because of the physical significance of the matrix $\mathbf{A}$, one expects that placing the matrix entries corresponding to tightly-coupled circuit nodes into M would improve the convergence of the band relaxation. In [Lumsdaine88], a heuristic algorithm is used that orders $\mathbf{A}$ to improve the convergence properties of the relaxation, while minimizing the bandwidth of $\mathbf{M}$. The heuristic used requires that a_{ij} and a_{ji} be in $\mathbf{M}$ if

$$\frac{\max^2(a_{ji}, a_{ij})}{|a_{ii}a_{jj}|} \geq \alpha, \tag{5.3}$$

where α is a parameter that is tuned for each circuit and adaptively changed based on its influence on the band-size produced.

The matrix $\mathbf{M}$ is derived by first constructing a graph based on $\mathbf{A}$ which has edges between pairs of nodes only if the pair of nodes is "tightly coupled". The so-constructed graph of the matrix $\mathbf{A}$ is reordered to minimize the bandwidth of $\mathbf{M}$ using the Reverse-Cuthill-McKee algorithm [Duff86, George81]. This produces an ordering for $\mathbf{A}$ which will insure that the tightly coupled equations lie together in an easily extracted band, i.e., in $\mathbf{M}$, and automatically produces the band-size for $\mathbf{M}$.

Algorithm 5.1 forms the core of the band-relaxation method. Given $\mathbf{x}, \mathbf{b} \in \Re^n$ and $\mathbf{A}, \mathbf{M} \in \Re^{n \times n}$, $\mathbf{M}$ being the band of tightly coupled elements contained in $\mathbf{A}$, this algorithm solves

$\mathbf{Ax} = \mathbf{b}$ iteratively for $\mathbf{x}$. Since the circuits being simulated are nonlinear, the values of the entries of $\mathbf{A}$ will change during the course of the simulation, possibly worsening the convergence characteristics of the band relaxation. It has been empirically observed that in those cases, it is best to reorder the matrix whenever the algorithm fails to converge, which may change the band-size.

Algorithm 5.1 (Variable Band-Relaxation)

variable_band_relaxation()
{
  given α *determine* $\mathbf{M}$
  $k = 0$
  $\mathbf{x}^0 = \mathbf{0}$
  repeat {
    $k = k + 1$;
    band-solve $\mathbf{Mx}^k = \mathbf{Nx}^{k-1} + \mathbf{b}$
    if $k >$ *max_iterations signal Newton non-convergence and reorder* $\mathbf{A}$
  }
  until $\left(\max_{1 \le i \le N} \left| x_i^k - x_i^{k-1} \right| \le \epsilon \right)$
}

We now note that this variable Band-Relaxation method can also be used in a nonlinear relaxation algorithm for the Newton method solution. In fact, if during the nonlinear relaxation, the relaxation is not continued until convergence, but instead only one relaxation iteration is performed per Newton iteration, then this algorithm is exactly similar as doing only one relaxation iteration on a linear relaxation method. Since the variable Band-Relaxation method has better convergence properties than Gauss-Jacobi in the linear case, one would expect that it would also accelerate nonlinear relaxation which is reported in [Lumsdaine88].

Results obtained with this algorithm were very promising since the speed of convergence has been greatly increased [Lumsdaine88], although some questions are still open as to whether such a method is clearly superior to block relaxation algorithms. The band-relaxation approach may, at some points, require a full matrix calculation before reordering the equations, and always implies the solution of a banded system. The computation of all matrix elements is not such a serious drawback since the elements can be obtained at virtually no additional cost by parallel device evaluation. The solution of the banded system can be a drawback since it is a more expensive operation, both in computational and communication terms, than device evaluation. However, in terms of number of iterations and overall convergence the results reported are

$$\begin{bmatrix} \mathbf{X} & \mathbf{X} & \mathbf{0} & \cdots & \cdots & \cdots & \cdots & \cdots & \mathbf{0} \\ \otimes & \mathbf{X} & \otimes & \mathbf{0} & \cdots & \cdots & \cdots & \cdots & \mathbf{0} \\ \mathbf{0} & \mathbf{X} & \mathbf{X} & \mathbf{X} & \mathbf{0} & \cdots & \cdots & \cdots & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \otimes & \mathbf{X} & \otimes & \mathbf{0} & \cdots & \cdots & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{X} & \mathbf{X} & \mathbf{X} & \mathbf{0} & \cdots & \mathbf{0} \\ \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \end{bmatrix} \longrightarrow \begin{bmatrix} \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \mathbf{0} \\ \mathbf{0} & \mathbf{X} & \mathbf{0} & \oplus & \cdots & \cdots & \cdots & \mathbf{0} \\ \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \mathbf{0} \\ \mathbf{0} & \oplus & \mathbf{0} & \mathbf{X} & \mathbf{0} & \oplus & \cdots & \mathbf{0} \\ \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \mathbf{0} \\ \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \end{bmatrix}$$

FIGURE 5-1: Illustration of a step of the Cyclic Reduction algorithm.

encouraging, especially when compared to Gauss-Jacobi relaxation.

5.2.2 A Parallel Band-matrix Solver

In this section we address the problem of solving the system of equations

$$\mathbf{M}\ \mathbf{x}^{k+1} = \mathbf{b}' \tag{5.4}$$

where $\mathbf{x}^{k+1}, \mathbf{b}' \in \Re^n$ and $\mathbf{M} \in \Re^{n \times n}$, $\mathbf{M}$ being a band matrix of bandwidth m, on a massively parallel computer.

We will assume that the band size is much less than n as the intent is to group together equations relating nodes with a tight coupling, as we saw in section 5.2.1. It is known that by using parallel cyclic reduction or nested dissection [Duff86], $\mathbf{M}$ can be factored in order $O(\log n)$ steps on a parallel processor with order $O(n)$ processors [Dongarra87].

Parallel Cyclic Reduction can be effectively implemented on the *Connection Machine*. The idea behind this algorithm is to recursively halve the size of the system by continually eliminating the even variables. If the size of the matrix is a power of two, this is efficiently implemented on the *Connection Machine* since communication can be minimized by setting all the odd rows to be nearest neighbors on the log(n) hypercube. Figure 5.2.2 illustrates the algorithm. In the figure, $\otimes$ represents a matrix element (corresponding to an even variable) while $\oplus$ represents a fill-in generated in the elimination process.

In terms of equations this algorithm can be depicted as follows. Suppose we start with the following block matrix:

$$\begin{Bmatrix} (x_1) & (x_2) & 0 & \cdots & \cdots & \cdots & \cdots & 0 \\ (x_1) & (x_2) & (x_3) & 0 & \cdots & \cdots & \cdots & 0 \\ 0 & (x_2) & (x_3) & (x_4) & 0 & \cdots & \cdots & 0 \\ 0 & 0 & (x_3) & (x_4) & (x_5) & 0 & \cdots & 0 \\ \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \end{Bmatrix}$$

Eliminating the odd variables one gets:

$$\begin{cases} (x_2) & (x_4) & 0 & \cdots & \cdots & \cdots & 0 \\ (x_2) & (x_4) & (x_6) & 0 & \cdots & \cdots & 0 \\ 0 & (x_4) & (x_6) & (x_8) & 0 & \cdots & 0 \end{cases}$$

Recursively following this procedure the system is finally reduced to a single equation which is solved. Backward substitution can then be used to determine the value of the other unknowns using a recursive procedure. The only drawback of this type of algorithm seems to be the fact that each time the system is halved, only half of the processors previously being used will become necessary for the next step. This leads to an overall waste of resources which may nevertheless be offset by the overall low computational requirements.

However, the algorithm pursued in the parallel implementation of a circuit simulator using nonlinear band-relaxation was based on an extension of an algorithm with different properties [Silveira88]. The algorithm described in this section is a generalization for block matrices of the algorithm described in [Wang81] which, although more expensive computationally, has very interesting communication requirements.

The general form of the algorithm is based on a partition of the matrix, perhaps augmenting its size, into P blocks of size k. Internally the whole matrix is divided in such a way that each $m \times m$ set of rows is in fact a group of three matrices, the a matrix composed on the $m \times m$ elements on the m-block diagonal, the sub-diagonal c matrix and the super-diagonal b matrix. In Figure 5-2 is a representation for a $5m \times 5m$ matrix (Each a, b, and c are $m \times m$ matrices).

a	b			
c	a	b		
	c	a	b	
		c	a	b
			c	a

FIGURE 5-2: Block Tridiagonal Matrix showing partitioned blocks.

This representation makes it easier to understand the algorithm that follows. The underlying goal is to eliminate the sub- and super-diagonal blocks (b's and c's) in order to get a block diagonal matrix which is then solved directly.

Below is a graphical interpretation of the 5 stages of the algorithm for a matrix with $P = 4$ and $k = 4$ (see Fig. (5-3)). Each element in the matrix should be considered as a block matrix $\in \Re^{m \times m}$, either an a, a b, or a c block.

The following steps are applied:

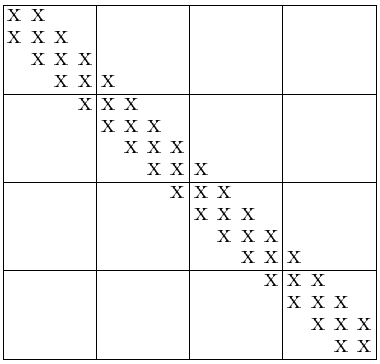

FIGURE 5-3: Example block tridiagonal matrix with 4 partitions of 4 row-blocks each.

Step 1 – *For each of the P partitions use the first block-row to eliminate the sub-diagonal entry of the block row below it. Do likewise, in a sequential fashion for block-rows $2, 3, \cdots, k-1$ of all partitions. These operations cause a fill-in to be produced in the last column of each sub-diagonal block (see Fig. (5-4)).*

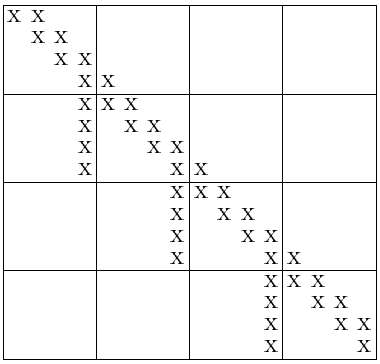

FIGURE 5-4: Example matrix after step 1 of Wang's algorithm.

Step 2 – *For each of the P partitions use the $k-1$-th block-row to eliminate the super-diagonal entry of the block row above it. Do likewise, in a sequential fashion for block-rows $k-2, \cdots, 2, 1$ of all partitions. These operations cause a fill-in to be produced in the last column of each diagonal block and in the last column of each super-diagonal block (see Fig. (5-5)).*

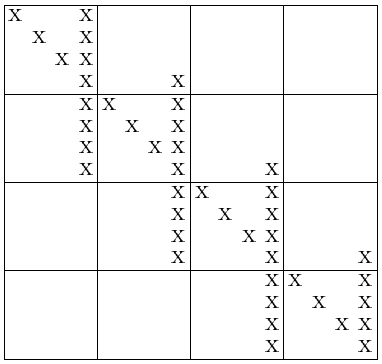

FIGURE 5-5: Example matrix after step 2 of Wang's algorithm.

Step 3 – *Use the last block-row of the first partition to eliminate the sub-diagonal fill-in of the last block row of the second partition. Do likewise in a sequential fashion for partitions* $2, 3, \cdots, P-1$ *(see Fig. (5-6)). Afterwards, for each of the* P *partitions, use the last block-row of each partition, to eliminate the sub-diagonal fill-in of the first block-row of the partition below it. Do likewise for block-rows* $2, 3, \cdots, k-1$ *of the partition below it (see Fig. (5-7)).*

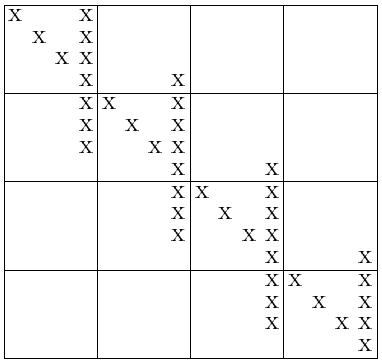

FIGURE 5-6: Example matrix after the first part of step 3 of Wang's algorithm.

Step 4 – *Use the last block-row of the last partition to eliminate the super-diagonal fill-in of the last block row of the previous partition. Do likewise in a sequential fashion for partitions* $P-1, \cdots, 2$ *(see Fig. (5-8)). Afterwards, for each of the* P *partitions, use the last block-row of each partition, to eliminate the super-diagonal fill-in of the first block-row of that partition. Do likewise for block-rows* $2, 3, \cdots, k-1$ *of that partition (see Fig. (5-9)).*

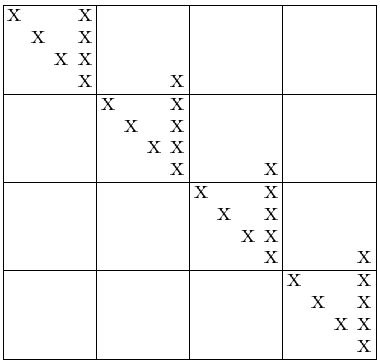

FIGURE 5-7: Example matrix after step 3 of Wang's algorithm.

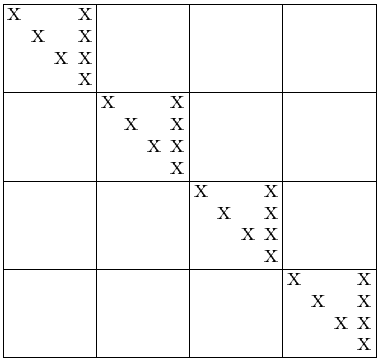

FIGURE 5-8: Example matrix after the first part of step 4 of Wang's algorithm.

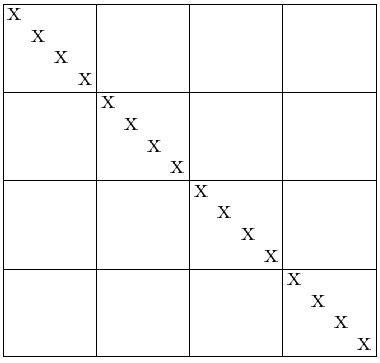

FIGURE 5-9: Example matrix after step 4 of Wang's algorithm.

Step 5 – *Solve the $m \times m$ block diagonal systems.*

As can be seen above, the final goal of obtaining a block diagonal matrix was achieved. In the figures the equations' right-hand-side was not depicted, but obviously it must be taken into consideration in order to produce an equivalent set of equations at each stage of the algorithm.

It is now worthwhile considering for a moment the computational complexity of this algorithm. In Table 5-1, the five stages of the algorithm are expressed in terms of their complexity. Note that since this version of the algorithm involves block matrices of size m, each step of each stage requires the factorization of an $m \times m$ matrix, the solution of an $m \times m$ systems of equations based on the factorized matrix, up to three $m \times m$ matrix products and additions. So the total number of steps is the one shown below times a constant factor (the sum of all individual steps just seen). However we will assume for our discussion, that the number of steps involving the above operations is the appropriate measure and correct it when necessary.

Stage	Number of Steps
1	P (k-1)
2	(P-1) (k-1)
3a	(P-1)
3b	(P-1) (k-1)
4a	(P-1)
4b	P (k-1)
5	P k
Total	5 P k - 2 P - 2 k

Table 5-1: Sequential Complexity of the Modified Wang algorithm.

As $Pk = \frac{n}{m}$, the algorithm described has a complexity of

$$5PK - 2P - 2k = O(\frac{n}{m}) \tag{5.5}$$

It is somewhat interesting to look at the values of P and k for which the number of sequential steps is minimized. Recalling that in our case, m is determined by physical considerations (since the band size is determined up to same point by the effective coupling of the circuit nodes), we are in fact interested in the function relating P and k such that the two values minimize the total number of steps.

Assuming a given P we know that $k = \frac{n}{mP}$

$$SS = 5Pk - 2P - 2k = 5\frac{n}{m} - 2\frac{n}{mk} - 2k\frac{\partial SS}{\partial k} = 2\frac{n}{mk^2} - 2 \tag{5.6}$$

the minimum is found for:

$$\frac{\partial SS}{\partial k} = 0 \quad \Rightarrow k = \sqrt{\frac{n}{m}} \tag{5.7}$$

So the optimum choices are:

$$\boxed{k = P = \sqrt{\frac{n}{m}}} \tag{5.8}$$

To conclude, Wang's algorithm, if implemented on a sequential machine, and provided m is independent of n, has a complexity of $O(n)$. This is not surprising, because a careful look tells us that the algorithm is nothing more than a simple, smart, two-way Gaussian elimination on the band elements.

We now consider the parallel complexity of the modified Wang algorithm. In Table 5-2, the five stages of the algorithm are expressed in terms of their complexity. Note that again, since this version of the algorithm involves block matrices of size m, each step of each stage requires the factorization of an $m \times m$ matrix, the solution of m $m \times m$ systems of equations based on the factorized matrix and, again, up to three $m \times m$ matrix products and additions. So the total number of steps is the one shown below times a constant factor (the sum of all individual steps just seen), which we will again ignore for the time being.

Stage	Parallelism	Number of Sequential Steps
1	P	k-1
2	P	k-1
3a	-	P-1
3b	P	k-1
4a	-	P-1
4b	P	k-1
5	P	1
Total	-	2 P + 4 k - 5

Table 5-2: Parallel Complexity of the Modified Wang algorithm.

Let us once again try to find the values P and k for which the number of parallel steps is minimized. This will tell us of the effective complexity of the algorithm. Remembering that m is determined by physical considerations, we will look for the relation between P and k such that the two values minimize the total number of steps. Assuming a given P we know that $k = \frac{n}{mP}$

$$PS = 2P + 4k - 5 = 2\frac{n}{mk} + 4k - 5 \quad \frac{\partial PS}{\partial k} = -2\frac{n}{mk^2} + 4 \tag{5.9}$$

the minimum is found for:

$$\frac{\partial PS}{\partial k} = 0 \quad \Rightarrow k = \sqrt{\frac{n}{2m}} \tag{5.10}$$

So the optimum choices are:

$$\boxed{k = \sqrt{\frac{n}{2m}}} \qquad \boxed{P = \sqrt{\frac{2n}{m}}} \tag{5.11}$$

With this values, then the complexity is at best:

$$PS = 2P + 4k - 5 = 2\sqrt{\frac{2n}{m}} + 4\sqrt{\frac{n}{2m}} - 5 \simeq 6\sqrt{2}\sqrt{\frac{n}{m}} = O(\sqrt{n}) \tag{5.12}$$

The above result tells us that that the Modified Wang algorithm has a complexity growing with $O(\sqrt{n})$ if implemented on a parallel machine. In fact, the total number of steps calculated is greater than the necessary number, since there is extra parallelism on stages $3b$ and $4b$ of the algorithm that we did not consider. However the total number of steps remains as $O(\sqrt{n})$.

We saw before that there are algorithms whose complexity only grows with $O(\log n)$. The decision to choose this algorithm was simply because it presents the nice feature of requiring few non-nearest neighbor communications. In fact, except for stages $3a$ and $4a$ all communications can be done on a nearest neighbor basis.

5.2.3 Non-Linear Band-Relaxation on the Connection Machine

The algorithm described in section 5.2.2 was implemented as the core of the Band-Relaxation circuit simulation version on the *Connection Machine*. A preliminary version of the simulator using Gauss-Jacobi was the testbed for debugging the device models code. A sequential version of the band-solver was first implemented on a sequential machine and later ported to a *Connection Machine* front-end, for development of the parallel version. The Band-Relaxation circuit simulator version implemented only contains a few simple device types such that the parallel algorithm can be tested. More complicated device models were left to a second stage of development.

The data structures used for the band-solver were chosen in the context of the circuit simulator implementation. This implies that the data structures used were by no means the most efficient for the band-solver. In fact, as suggested in [Leighton88] there are data structures that can lend to more efficient parallel solution methods. Even though the matrix solution process itself dominates the total simulation time, care must be used in the matching of data to processors because the matrix has to be filled in and communication is expensive.

We will not debate here the constraints implied by filling the matrix, choosing the timestep, etc., but simply to explain how the matrix problem was solved. Our goal in the implementation was to have an easy to solve band-matrix structure, such that communication between matrix elements and devices would be somewhat minimized. We decided to use a bi-dimensional structure, consisting of matrix rows and devices. The matrix elements were placed in rows and n *Connection Machine* processors were dedicated to hold the data for those rows, i.e. one row per processor. These n processors were placed in a linear fashion (for simplicity we will call this "dimension 0"), such that each group of m consecutive processors would represent an $m \times m$ block matrix. However incorrect, we will refer to these processors as the *node* processors. The other dimension of the whole structure, which we will refer to as "dimension 1", consists of the devices.

In the first stage of this implementation only two terminal devices were included in the simulator. Although this is a constraint on the types of circuit that can be simulated, it is by no means a fundamental constraint regarding the problem at study: band-matrix solution. In fact, the structure implemented can contain multi-terminal devices without any change. Each device connected to a given node, say node k, is placed along "dimension 1" connected to the processor holding the data for the k^{th} row of the matrix. All other nodes that are also terminals for that device would, on their list of devices, a fake device, called fan-out device, that would actually perform as an actual device. When one of the device's terminal is ground, that terminal is obviously omitted as for any power terminal.

Therefore, we have a two-dimensional structure, with the matrix rows on one dimension and the devices, attached to its corresponding terminal nodes on the other dimension. This two-dimensional structure, and especially the choice of having each row of the matrix stored in the memory of a single *Connection Machine* processor, is certainly not the best way of solving the problem in terms of load balancing, since when the matrix is being solved, the device processors are idle, and when the devices are performing their computations, the row elements are idle. However, from the point of view of solving the matrix, this ensures that we minimize the communication between processors since now each processor has local access to a complete row of the matrix. And since most of the communication on the Modified Wang Algorithm is made between clusters of m processors (processors holding an $m \times m$ matrix, therefore m rows) this means that the bulk of communication is between processors a distance $O(m)$ apart.

Figure 5-10 gives a view of how the data structure is actually used. Each of the row or devices elements shown (Node1, $\cdots$, Node4, Dev11, $\cdots$, Dev44, etc.) is a *Connection Machine* processor. Each of the *node* processors holds the necessary data associated with a row of the full matrix. The device processors hold the necessary data to perform their calculation. Note that the strategy followed of having each processor hold one row of the matrix, i.e., a different node in the circuit, makes it very simple to implement a Gauss-Jacobi like scheme to solve the equations at each time step. This was the strategy followed by Don Webber; we used it for performance and convergence comparisons.

The simulation control algorithm stems from the data structure shown above. At each time step, the front-end determines the next time step to be used based on estimates on the local truncation error. Each processor then computes a guess for the next voltage step based on the last few values it has stored in its memory. On the first step, that guess is scanned out into the device processors. This implies a scan operation along "dimension 1" of the laid out structure. In the next step, each device processor that is not a fan-out asks its fan-outs for the other terminal voltages it needs. This is a global communication step. Once that information is passed along, all device processors have consistent values for their terminals voltages and evaluate their associated constituent equations and calculate the appropriate conductance, capacitance, charge and current. In the next step, these values are sent back to the fan-out devices. In the final

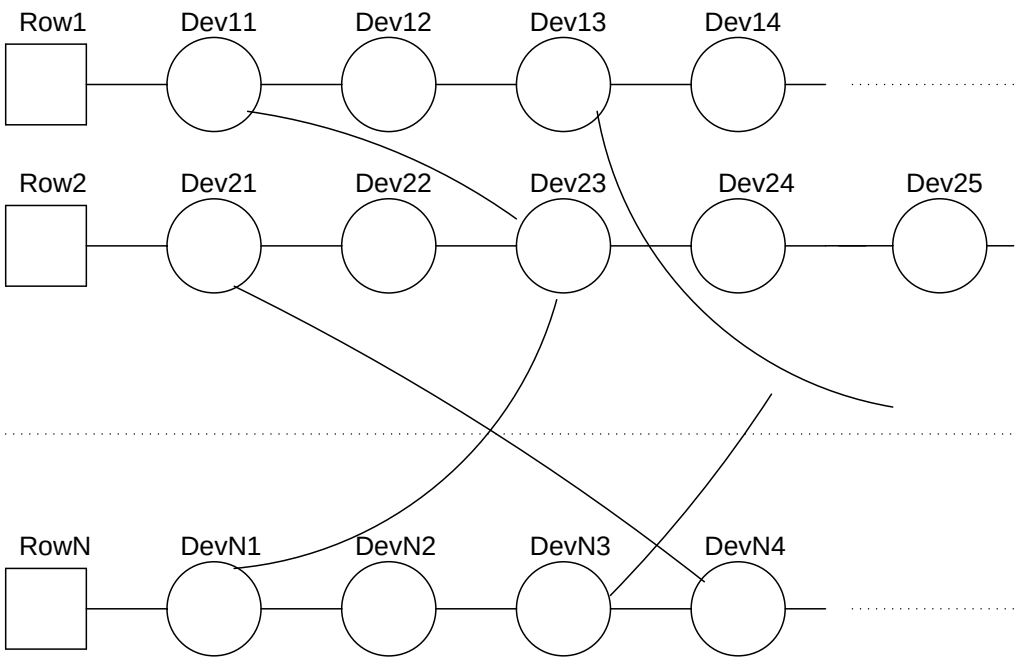

Figure 5-10: Data-Structures used in the Band-Relaxation Circuit Simulator implementation

step they are scanned back, along the axis 1, into the correct matrix positions.

After all the matrix elements are in place, the band-solver is called upon to solve the band matrix system. The final result is the vector of new node voltages, which can then be read in order to determine the status of the convergence of the method. If one were to do Gauss-Jacobi instead of the Band-Relaxation algorithm, the previous steps would be executed as stated, except the solution of the system of equations would be done in place, on an equation by equation basis, as is characteristic of the Gauss-Jacobi algorithm.

This flexibility allowed us to implement both methods and establish some comparisons. Some results will be presented ahead. Algorithm 5.2 is a pseudo-code description of the overall implementation.

Each row processor contains all the data associated with one matrix row, including space for the fill-ins above and under the diagonal. The elimination of an $m \times m$ block matrix, involves the LU decomposition of another matrix, the solution of a system of the form:

$$x^T M = b^T,$$

and up to three multiplications and additions of two $m \times m$ matrices. The algorithm was optimized such that every LU decomposition is done in place and each matrix is decomposed only once. It turns out that only the $m \times m$ diagonal block matrices have to be decomposed and, once that happens, they are not involved in any other elimination, being used only in the solution of the overall diagonal system obtained after the last stage of the algorithm, for which they are already decomposed. This was established after a detailed and careful study of the

Algorithm 5.2 (Parallel Nonlinear Band-Relaxation Circuit Simulation)

nonlinear_band_relaxation()
{
  $t = 0$;
  while *(t < Simulation_Time)* {
    choose h ;
    repeat { */* until convergence */*
      do parallel {
        guess the new node voltage
        scan the guess value out to the device processors
        get other terminal voltages from the fan-out devices
        use the constituent equations of the device to calculate capacitance, conductance and current for the element
        scan values into the processors holding the matrix rows and store them in the correct matrix positions
      }
      do parallel {
        solve the banded system of equations (for Band-Relaxation) and compute the new node voltages or
        compute the new node voltages locally (if using Gauss-Jacobi)
      }
    }
    until $\left(\max_{1 \le i \le N} \left| x_i^k - x_i^{k-1} \right| \le \epsilon \right)$
  }
}

algorithm on the sequential implementation. and it clearly minimizes the number of operations necessary.

The strategy followed at each stage, while dealing with an $m \times m$ matrix, was to have only one row active at a time (therefore only out of every m processors is active), and having that row fetch the necessary data to perform any operation. This implies that we do not take advantage of all the parallelism available at certain times, but ensures that the routing of data between communicating processors turns into a $one-to-one$ problem which certainly decreases the total communication time. Although this decision was not based on any behavior study, it seems that on the overall, the choice turns out to be a good one, since many times (for instance on LU-decomposing) only one row of a matrix can be dealt at each time, and if we had several rows active, we could have $many-to-1$ routing problems which will significantly increase the communication time due to packet collisions. Furthermore, some care was taken to minimize the number of reads from other processors, by making at least half of the necessary data for each operation reside on the active processor local memory. As an example, when multiplying matrices, the row being used is in the local memory of the active processor and only the corresponding column elements have to be fetched.

The underlying idea in all the stages of the Band-Solver algorithm is to activate all row processors. Then only the processors that contain data belonging to a specific k-row remain active for the elimination step that follows. In that elimination step, we activate all of the m rows in order, and deactivate them in sequence. This requires saving the context-flag of all processors every time we activate a new set, and the recovery of the processors status at the end of the operation. This is performed by means of a stack and some "push_context" and "pop_context" operations.

Since the code, being written in *Paris*, is by itself not very clear and straightforward, every stage of the algorithm is performed by a different function, that calls the general LU-decomposition routine, then determines the solution of the given m $m \times m$ systems (by backward and forward substitution), and adds the result to the correct place. This is done in three steps. First the P pivot matrices are decomposed, in parallel, and left in place. Then the multiple system solution is found, and a matrix multiplication and addition is done using in place results with all the elements in the row where the element to be eliminated is. Algorithm 5.3 is a pseudo-code description of actions performed in a general band-matrix such as the one present in Figure. 5-2.

The algorithm as described, implies that *gets*, rather than *sends* were used in the communication process. In vague terms this by itself is responsible for a performance degradation by a factor of two as compared to similar behavior using *sends*. This is nonetheless a second order factor in this project, and we simply will neglect it.

Algorithm 5.3 (Parallel Solution of Band Matrices)

solve_band_matrix()
{
LU-decompose the P diagonal blocks a_i, and keep them in place;
solve the right system $x^T a_{i_{LU}} = b_{i-1}$ and leave the result in place in an auxiliary variable;
form the products involving the sub-, super-diagonal lower fill-in blocks and the block resulting from the previous solution;
actualize the i-th block row (a block row is a row of $m \times m$ matrices) by subtracting the above products from the diagonal block and the lower and upper fill-ins above.
}

5.2.4 Discussion of Results

We will now present and discuss some results obtained with a stand-alone version of the band-solver as well as the results obtained with the Circuit Simulator when using the band-solver.

In order to test the performance of the band solver, we generated several random matrices, with different choices of n, P and k, therefore with different band-sizes, and tested them on the sequential and parallel implementations. The sequential machine is a conventional VAX 8800 which was also used as front-end for the *Connection Machine* system. The *Connection Machine* used had 8192 physical processors and therefore a virtual-to-physical processor ratio of 1 in all examples shown. Larger examples were not tried due to memory shortage on the machine. Table 5-3 groups the results we obtained, with a simple description of each of the matrices used.

Example	n	P	k	m	Time (Par) (sec)	Time (Seq) (sec)
opt4950	4950	45	22	5	100.03	18,42
opt1000	1000	20	10	5	43.05	3.55
opt490	490	14	7	5	29.10	1.67
x490	490	7	14	5	42.15	1.68
x1000	1000	25	8	5	43.16	3.5
y1000	1000	10	20	5	64.03	3.62
z1000	1000	20	5	10	223.58	11.15
w1000	1000	20	25	2	6.56	1.03
a4970	4970	71	35	2	12.5	5.23

Table 5-3: Performance Results for the Band-Solver Algorithm.

Note that the examples named *opt∗* correspond to optimal choices of P and k, given n and m.

It is clear from the above results that the performance of the Band-Solver is far from acceptable and certainly far from our expectations. However, a few interesting results can be drawn from the above figures. First, as we expect, we see that the sequential version of the Modified Wang Algorithm grows linearly with n on the examples shown, independent of the choice of P and k: *opt1000* is approximately twice as big as *opt490* and takes twice as much; also *opt4950* is 10 times as big as *opt490* an takes 10 times more; examples *opt1000*, *x1000* and *y1000* have similar running times, since n and m are kept in all these examples. On the Parallel implementation results, we also see a growth linear with $O(\sqrt{n})$ and the derived optimal choice for P and k do provide the best results (see *opt4950*, *opt1000* and *opt490* to verify the complexity growing as $O(\sqrt{n})$ and look for *y1000* to see how a variation on P and k away from the derived optimal values can produce much worse results).

Some care is needed while examining the above results. Note that for instances *z1000* shows a large running time (both on the sequential and parallel version), while *w1000* or *a4970* are solved very quickly. All these examples present different values of m. To fully understand the meaning of the results presented, we must remember that the derived complexity results were precise up to a multiplicative factor. That factor took into account the necessary operations for decomposing an $m \times m$ matrix and multiplying and adding $m \times m$ matrices. This has a complexity of its own that is roughly $O(m^3)$, and therefore the complexity of the overall sequential algorithm turns out to be at least on the order of $O(m^2 n)$ (and not $O(n)$ as we said before). The parallel algorithm is also on the order of $O(m^{\frac{5}{2}}\sqrt{n})$. This detail is however only interesting to understand the above results in greater depth, and presents no added information of relevance.

It is also important in the sense that the routines performing these functions are not easily parallelizable at the level we were pursuing. A finer-grain parallelism approach is necessary to fully exploit a greater degree of parallelism in those sub-problems that become computationally very expensive as m grow. This question however, was not studied in depth. It is important nonetheless, to remember that in our application m is a function of the circuit, therefore not a completely free parameter. Hence, we may end up with large values of m on certain circuits, therefore greatly increasing the run time of the simulator.

The results in Table 5-3 predict a break-even point for the parallel implementation that is far beyond our capacity to test at the moment due to disk space limitations for storing the matrices! This break-even point is obviously dependent on m and grows with m. Another meaningful measurement to be obtained is the number of communication steps used throughout the algorithm. This can be approximately measured by counting the number of gets on a typical run. For the optimal choice examples we measured the values presented in Table 5-4.

Using a 1 mili-second value for an average communication step and taking into account that

Example	# of gets	Communication time (sec)
opt4950	47085	94.170
opt1000	20280	40.560
opt490	13695	27.390

Table 5-4: Number of reads from other processor's memory.

a *get* takes approximately twice as much as a *send*, we get a communication burden as shown in Table 5-4. Although these values are somewhat of a worst case scenario, they in fact tell us something about the performance of the machine and the implementation of the algorithm. Interestingly enough, the values depicted in Table 5-4 also grow as $O(\sqrt{n})$, which is as expected.

Table 5-5 presents some results obtained with the circuit simulator using a sequential Band-Solver, a Gauss-Jacobi Method and the parallel implementation of the Modified Wang Band-Solver algorithm. The test circuit were simple RC trees with increasing number of nodes and devices.

Example	# of devices	CPU Time (seconds)			
		Band-Solver	GJ (CM)	Seq. Band	GJ (FE)
m16	31	33	404	1.7	32
m32	63	49	382	3.2	72
m600	1199	176	441	60	1320
ext1200	-	250[1]	-	120[1]	-
ext2400	-	350[1]	-	240[1]	-
ext4800	-	500[1]	-	480[1]	-

Table 5-5: Non-Linear Band-Relaxation Circuit Simulation results.

The results shown are somewhat encouraging in terms of circuit simulation. In fact, the parallel version on the *Connection Machine* was able to produce better results than the Gauss-Jacobi sequential version (as expected) and a break-even point for the Band-Relaxation Non-Linear Relaxation was in sight for circuits with a reasonably large number of nodes. Once more the results followed the expected patterns for the sequential and parallel version, respectively $O(n)$ and $O(\sqrt{n})$, while the Gauss-Jacobi version has a constant running time.

The results presented show that efficient Non-Linear Band-Relaxation for moderately to large circuits is possible in a parallel architecture. However, from the results presented in the previous section we cannot infer a decisive answer to the question of whether a massively parallel computer is best for solving band matrices, and therefore for solving the circuit simulation problem. Run-time profiles of simulations have shown that up to 20% of the total simulation

[1] Predicted results based on extrapolation of previous values

time can be spent on an idle sequence, waiting for data to arrive at the destination, and up to 75% of the time is spent on *Connection Machine* communications related functions, initialization procedures and security checks.

Another important issue relates to the algorithm being used to solve the band matrix in parallel. The Modified Wang Algorithm presented here, is not necessarily the best possible option. Other methods, discussed above, like Block Cyclic Reduction [Duff86] present a complexity growing with $O(m^3 + m^3 \log(\frac{n}{m})$, although they are more communication intensive. Also, other methods are known [Dongarra87, Hockney88] which promise complexities of the same order.

This first parallel implementation allowed for a much better understanding of the architectural features of the machine. Webber's implementation, for instance, simply setup a linear list of processors and devices, therefore not taking much advantage of the physical underlying structure of the machine: a 12 dimensional hypercube [Hillis85, Hockney88, Seitz85, Leighton88, Wiley87]. Our implementation was two-dimensional with matrix rows in one dimension and devices in the other which provided a more efficient usage of the physical channels for communication, therefore reducing the communication time. The proposed block cyclic reduction layout of data to processors could provide an even better match for the *Connection Machine* architecture.

If one takes into account the amount of raw computational devices, the speedups obtained is much less than expected. We note that the circuits for which the parallel Band-Relaxation circuit was able to outperform the sequential version were too large for the memory available. We point however, that for circuits containing devices with complex models some additional speedup can be expected.

Based on the above presented results it seemed at this point that in order to ascertain whether massively parallel circuit simulation was a worthwhile effort to pursue, we would have to improve on some of the disadvantages reported. Namely:

- Increase the size of the test problems,
- Study other linear solvers such as parallel cyclic reduction,
- Increase the parallelism inherent in the problem,
- Restrict our study to more regular types of circuitry.

The first possibility is only academically useful since usual circuits are restricted in size either by fabrication or design constraints. The second possibility seems more interesting. However, cyclic reduction has been successfully tested elsewhere, and an implementation here would not shed any light as to whether the circuit simulation problem is indeed well suited for these types of architectures.

The third possibility was already studied in section 3.3 and the conclusions drawn were not very enthusiastic. At this point it is therefore reasonable to state our belief that generic circuit

simulation does not present good characteristics for implementation on a massively parallel architecture. A circuit can be very irregular, which makes it more difficult to simulate due to the irregular communication patterns derived. Also we believe that not enough parallelism is generated in the usual algorithms for the solution of the equations describing the electrical behavior of the circuit. In section 5.3 we will restrict ourselves to a class of circuit that meets the pointed requirements of size and regularity and show that for those types of circuits, extremely efficient simulation can be performed on the massively parallel computer architecture.

5.3 Simulating Analog Grids on the Connection Machine

The task of any vision system is to reconstruct information about the three-dimensional world using a two-dimensional image of that world. The problems which such a system must solve therefore tend to be ill-posed. These problems are typically solved by imposing appropriate constraints and solving the resulting minimization problem [Horn86]. The computational costs involved in numerical implementations of vision algorithms are large. Although efficient computation techniques exist for numerically computing the solutions to the regularization problems [Terzopoulos84], even the fastest algorithms running on a parallel supercomputer (such as the *Connection Machine*) do not even begin to approach a level of performance that one could call "real time." However the solutions generally have the property that they can be locally computed in parallel and are therefore amenable to (VLSI) network implementation [Koch85, Poggio84, Horn88, Hutchinson88]. Because of the potential cost and performance benefits, VLSI implementation of vision algorithms is a topic of much current research [Wyatt88].

Marr [Marr82] separates vision tasks into three general stages:

- Early Vision, in which the tasks are to detect shape, depth, discontinuities, etc., from raw data;
- Intermediate Vision, in which the tasks are to reconstruct surfaces in a viewer-centered coordinate frame; and
- Late Vision, in which the tasks include reconstruction of the three-dimensional world in the world coordinate system and other high-level tasks such as object recognition.

Early vision is the most low-level stage, involving the initial stages of image processing. Intermediate vision deals with the integration of early vision modules. Late vision involves tasks which are best handled by Artificial Intelligence techniques.

Many problems in early vision can be formulated in terms of minimizing an energy or cost function. Examples are shape-from-shading, edge detection, motion analysis, structure from motion, surface interpolation and image smoothing and segmentation [Poggio85, Lumsdaine89,

Lumsdaine90]. In some cases, these filtering tasks can be accomplished using one and two dimensional resistive grids [Mead88], a fact that has sparked interest in general analog signal processors based on grid connections of components.

5.3.1 Circuits for Early Vision Tasks

These so-called "vision circuits" form a class of circuits which, for the most part, cannot be simulated with traditional programs. These circuits are necessarily very large and must be simulated with high accuracy, i.e. at the analog level. Some of the most ambitious circuits (such as those planned for the MIT vision chip project [Wyatt88]) require solving for *hundreds of thousands* of unknowns. Because of their sheer size, these circuits are very expensive to simulate and are rarely or never simulated in their entirety. Typical analog circuit simulators running on conventional computers are not able to handle them simply because of their immense size, since the computation time for these simulators grows super-linearly with the size of the circuit.

The general form of the vision circuit is that of a large, regular array of analog circuit cells. Variations come in the form of the tasks to be performed, the circuit components, and the interconnections between components. In the following discussion, many of the circuits are grids of cells. When we refer to the size of a grid, we are referring to the number of cells in each dimension, not to the size of the circuit, since each cell may be composed of multiple elements. Perhaps the simplest, but most useful, vision circuit is a grid of linear resistors. A grid of linear resistors is useful in vision for accomplishing such tasks as spatial convolution or interpolation. This circuit has the same form as a discrete approximation to a resistive sheet, which in turn has some nice properties. For instance, the potential distribution on a resistive sheet satisfies Laplace's equation. Simulating a grid of linear resistors is not such a difficult problem in itself. However, the resistive grid, as simple as it may be, still conveys most of the problems that these type of circuits present and is an ideal starting point for the solution of the problems we investigate.

Consider an $N \times N$ grid of nodes having resistors of resistance r between neighboring pairs and a circuit consisting of a conductance g in series with a constant voltage source of value u to ground at each node, as shown in Fig. 5-11.

Let $\mathbf{J} \in \Re^{N^2}$ be a vector of currents injected at the circuit nodes, which for this case is found to be, for node i:

$$J_i = \frac{u_i}{g}.$$

The vector of node voltages, $\mathbf{v} \in \Re^{N^2}$, is the solution to the equation

$$\mathbf{G}\,\mathbf{v} = \mathbf{J}. \tag{5.13}$$

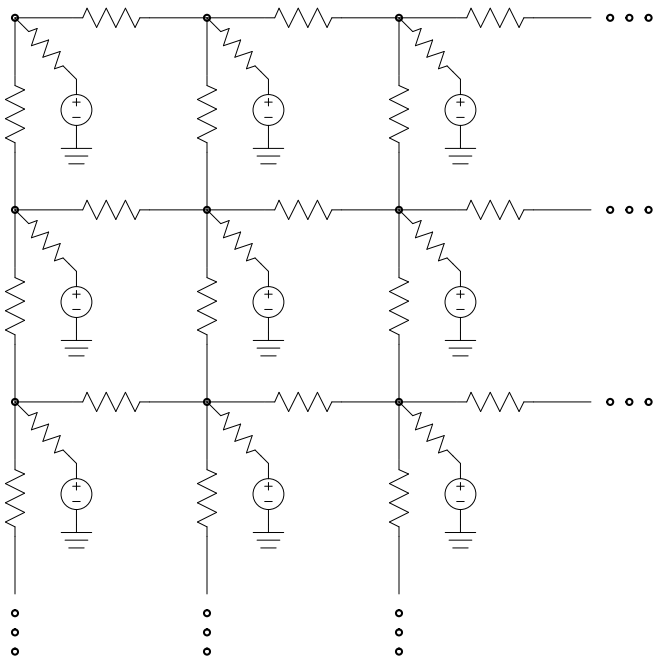

FIGURE 5-11: Grid of Linear Resistors.

Here, $\mathbf{G} \in \Re^{N^2 \times N^2}$ is defined to be

$$\mathbf{H} = \frac{1}{r} \begin{bmatrix} (\mathbf{P} - \mathbf{I}_N) & -\mathbf{I}_N & 0 & \cdots & 0 \\ -\mathbf{I}_N & \mathbf{P} & -\mathbf{I}_N & \cdots & 0 \\ 0 & & \cdots & & 0 \\ 0 & \cdots & -\mathbf{I}_N & \mathbf{P} & -\mathbf{I}_N \\ 0 & \cdots & 0 & -\mathbf{I}_N & (\mathbf{P} - \mathbf{I}_N) \end{bmatrix} + g\, \mathbf{I}_{N^2}, \tag{5.14}$$

where $\mathbf{P} \in \Re^{N \times N}$ is given by

$$\mathbf{P} = \begin{bmatrix} 3 & -1 & 0 & \cdots & 0 \\ -1 & 4 & -1 & \cdots & 0 \\ 0 & & \cdots & & 0 \\ 0 & \cdots & -1 & 4 & -1 \\ 0 & \cdots & 0 & -1 & 3 \end{bmatrix}. \tag{5.15}$$

If the circuit of Fig. 5-11 is modified by adding a capacitor with capacitance c to ground at each node, the circuit shown in Fig. 5-12 is obtained.

The dynamics of this circuit are then of the form of equation (2.1), or more specifically,

$$\mathbf{C}\, \frac{d}{dt}\mathbf{v} + \mathbf{G}\, \mathbf{v} = \mathbf{J}. \tag{5.16}$$

If the trapezoidal rule is applied to discretize this ODE, we obtain, at time t_{m+1}

$$(\mathbf{C} + \frac{h}{2}\mathbf{G})\mathbf{v}(t_{m+1}) = (\mathbf{C} - \frac{h}{2}\mathbf{G})\mathbf{v}(t_m) + \frac{h}{2}(\mathbf{J}(t_{m+1}) + \mathbf{J}(t_m)), \tag{5.17}$$

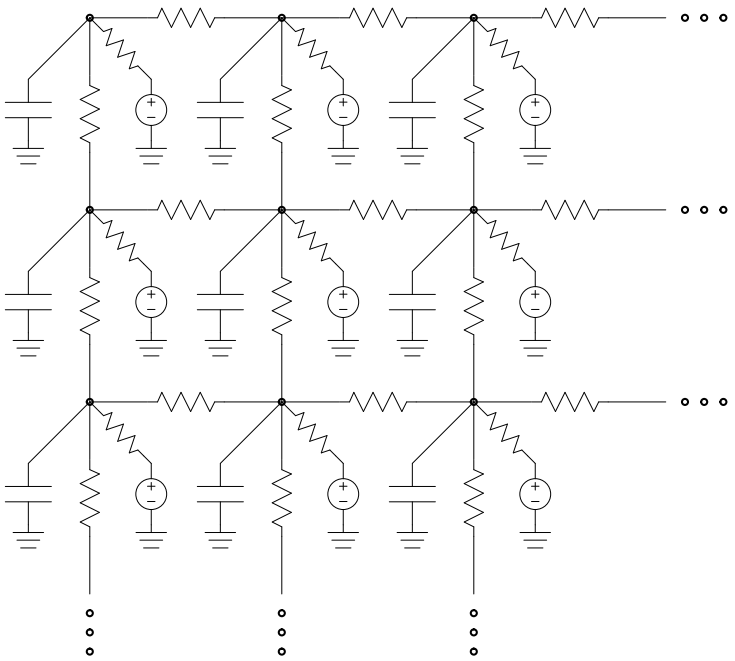

Figure 5-12: Grid of Linear Resistors with capacitor to ground at each node.

where $\mathbf{G}$, and $\mathbf{J}$ are given above, and $\mathbf{C} \in \Re^{N^2 \times N^2}$ is given by

$$\mathbf{C} = c\, \mathbf{I}_{N^2}. \tag{5.18}$$

The formulation of the static and dynamic problem for the grid of linear resistors formulated above in Eqn. (5.13) and Eqn. (5.16) are very similar to two problems which are well-studied — namely Poisson's equation and the heat equation — and therefore the work involved in developing simulation tools for this problem is essentially a translation of PDE techniques into the circuit domain. However, we recall that since the size of the grids can be very large ($N = 128$ or larger), the amount of necessary computations is prohibitive for todays computers. The question to be addressed is how to exploit the regularity of the circuit to efficiently simulate this circuit on a machine with the architectural characteristics of the *Connection Machine*.

A somewhat more general form of the linear resistive grid is the idealized two-dimensional nonlinear resistive grid used for image smoothing and segmentation [Lumsdaine89], as shown in Fig. 5-13.

In this example, the circuit elements are chosen to implement a certain signal-processing task (i.e., image smoothing and segmentation). Obviously, in practice, such a circuit would be built in analog VLSI using semiconductor devices to implement this functionality. The idealized network is useful for illustrating the salient points of the simulation system requirements, and is therefore studied here for purposes of discussion. Note however that all discussion of the idealized network applies to its physical VLSI implementation as well. The simulation of actual VLSI circuits will in fact present some advantages when considering the implementation of the

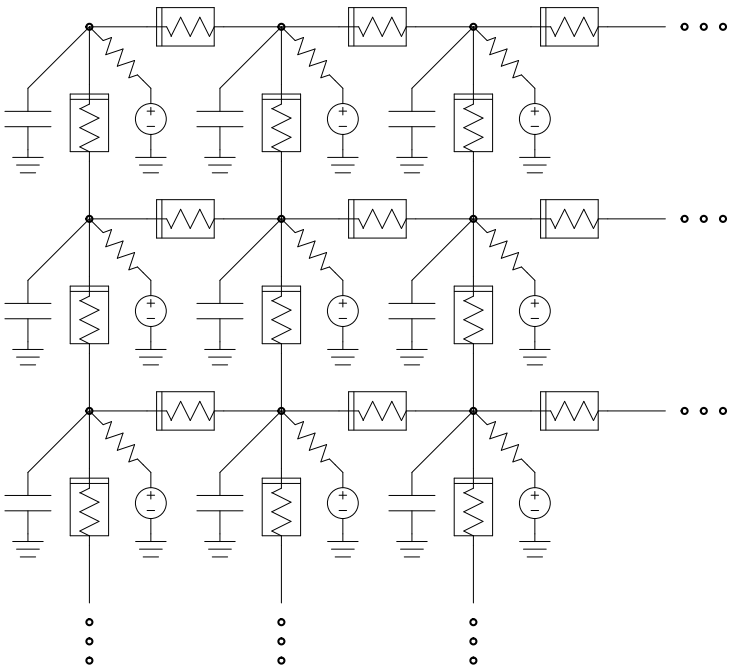

FIGURE 5-13: Grid of nonlinear resistors with capacitor to ground.

simulator on the *Connection Machine*, as will be discussed later.

To determine the dynamics of this circuit, we formulate the node equations for a grid point i, j in the network as

$$\begin{aligned} c_i \tfrac{d}{dt} v_{i,j} = g(v_{i,j} - u_{i,j}) + i_\beta(v_{i,j} - v_{i+1,j}) \\ +i_\beta(v_{i,j} - v_{i-1,j}) + i_\beta(v_{i,j} - v_{i,j+1}) + i_\beta(v_{i,j} - v_{i,j-1}) \end{aligned} \tag{5.19}$$

where $v_{i,j}$ is the node voltage and $u_{i,j}$ represents the voltage source to ground at each node, g is the input source impedance, c is the parasitic capacitance from the grid node to ground, and $i_\beta(.)$ is a nonlinear "fused resistor" whose constitutive relation is given by

$$i_\beta(v) = \frac{\lambda_s v}{1 + e^{-\beta[\lambda_l - \lambda_s v^2]}}, \tag{5.20}$$

where λ_s and λ_l are parameters whose values depend on the type of operation to be performed.

For a smoothing and segmentation task, the value of the voltage source to ground at each node would be the image input at the grid point i, j Similarly, the input source impedance g, can be used to control the amount of smoothing to be exercised. The parameter β alters how abruptly the resistor current reduces with increasing voltage, and is often made time-varying to perform certain types of continuations. The i-v relation for this element is illustrated in Fig. 5-14 for several values of β.

The system of differential equations that describes a grid of size N by N can be generated by replication of Eqn. (5.19) over the range of grid points $i, j \in 1, ..., N$. We describe the resulting

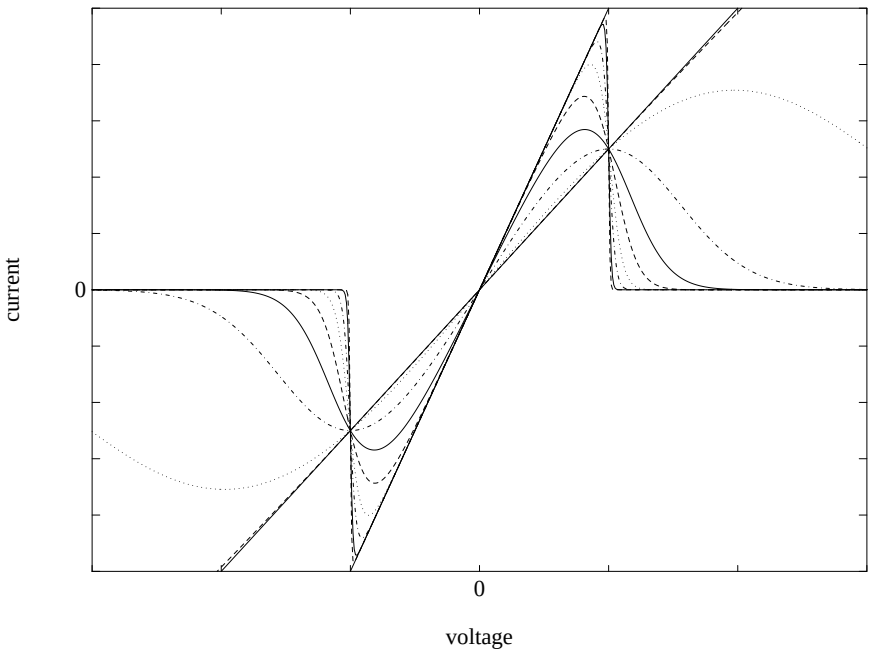

Figure 5-14: Nonlinear resistor i-v Characteristic.

system compactly as

$$\mathbf{C}\frac{d}{dt}\mathbf{v}(t) + \mathbf{f}\left(\mathbf{v}(t), \beta(t)\right) = \mathbf{0}, \tag{5.21}$$

where $\mathbf{v}(t) \in \Re^N$ is again the vector of node voltages, $\mathbf{f} : \Re^{N+1} \rightarrow \Re^N$ is a nonlinear function of $\mathbf{v}$ representing the resistive constitutive relations of the circuit, $\beta(t) \in \Re$ is the parameter of nonlinear resistors which is allowed to be a function of time.

Numerically integrating with the trapezoidal rule we get

$$\mathbf{v}(t_{m+1}) = \mathbf{v}(t_m) + \frac{h}{2}\,\mathbf{C}^{-1}\,\left[f\left(\mathbf{v}(t), \beta(t)\right) + \mathbf{f}\left(v(t), \beta(t)\right)\right] = \mathbf{0}. \tag{5.22}$$

This algebraic problem can be solved with Newton's method yielding at iteration k,

$$\mathbf{J}_F\left(\mathbf{v}^k(t_{m+1})\right)\left[\mathbf{v}^{k+1}(t_{m+1}) - \mathbf{v}^k(t_{m+1})\right] = -\mathbf{F}\left(\mathbf{v}^m(t_{m+1})\right) \tag{5.23}$$

where

$$\mathbf{F}\left(\mathbf{v}(t_{m+1})\right) = \frac{2}{h}\,\mathbf{C}\,\left[\mathbf{v}(t_{m+1}) - \mathbf{v}(t_m)\right] + \mathbf{f}\left(\mathbf{v}(t_{m+1}), \beta(t_{m+1})\right) + \mathbf{f}\left(\mathbf{v}(t_m), \beta(t_m)\right), \tag{5.24}$$

and the Jacobian $\mathbf{J}_F(v(t))$ is

$$\mathbf{J}_F\left(\mathbf{v}(t_{m+1})\right) = \frac{2}{h}\,\mathbf{C} + \frac{\partial \mathbf{f}\left(\mathbf{v}(t_{m+1}), \beta(t_{m+1})\right)}{\partial \mathbf{v}} \tag{5.25}$$

It is important to note that the grid structure of the network is kept, and therefore one can expect some structure to persist in the Jacobian matrix. This is easily seen in Eqn. (5.19) by

noting that the node equation at the grid point i, j in the network only involves the voltages values of the neighboring nodes. Therefore the Jacobian matrix in Eqn. (5.23) is structurally similar to the conductance matrix found for the linear case and given in Eqn. (5.14).

Although the linear equation system (5.23) can be solved by some form of sparse Gaussian elimination, this is not the most efficient technique for the type of matrices generated by the "fused resistor" grid. For this problem, $\mathbf{J}_F$ is symmetric, although because the nonlinear resistors can have a negative conductance, $\mathbf{J}_F$ is not always diagonally dominant. This, along with the symmetry, suggests a conjugate-gradient type solution algorithm given, for completeness, in algorithm 5.4 [Golub83]. Note that although $\frac{\partial \mathbf{f}}{\partial \mathbf{v}}$ can suddenly become ill-conditioned, $\mathbf{J}_F$ can avoid ill-conditioning by appropriate choice of timestep. See Theorem 5.1.

Algorithm 5.4 (Conjugate Gradient Algorithm for solving A x = b)

```
cg()
{
  x = 0
  r = b
  p = r
  ρ1 = ⟨r, r⟩
  k = 1
  while (ρ1 ≤ ε) {
    if k > 1 {
      β = ρ1/ρ2
      p = r + β p
    }
    w = A p
    α = ρ1/⟨p, w⟩
    x = x + α p
    r = r − α w
    ρ2 = ρ1
    ρ1 = ⟨r, r⟩
    k = k + 1
  }
}
```

Furthermore, the CG algorithm is in fact better suited as a solution method for this problem than sparse Gaussian-elimination. For once Gaussian-elimination has a complexity of $O(n^{\frac{3}{2}}) = O(N^3)$ (easily derived from the structure of the problem), whereas with certain mild assumptions, CG can be shown to have a complexity of only $O(n) = O(N^2)$. In fact,

each iteration of the algorithm requires only a matrix-vector product and two scalar products, i.e. $O(n) = O(N^2)$ operations, and as shown in Theorems 5.1 and 5.2, convergence of the CG iteration can be achieved with $O(1)$ iterations. Also, CG is more suited for the type of parallel implementation pursued and it is possible to arbitrarily improve the convergence rate of CG during transient simulation (see Theorem 5.1).

The reason why CG is more suited for a parallel implementation on the *Connection Machine* will be debated later when the implementation is described. That CG is faster than Gaussian elimination was demonstrated by simulating a 64×64 grid on a sequential computer, showing that CG is approximately three times faster.

CONJUGATE GRADIENT TRANSIENT SIMULATION OF ANALOG GRIDS

Theorem 5.1 *Let the Conjugate-Gradient algorithm be used to solve for the linear system of equations in Eqn. (5.23) generated at each Newton iteration. If there is a capacitor with value c, to ground at each grid node and no other capacitive element in the circuit, then for any $\epsilon > 0$, there exists h_m such that for all $h < h_m$ we have*

$$\frac{\|\mathbf{e}^{k+1}\|_{\mathbf{A}}}{\|\mathbf{e}^{k}\|_{\mathbf{A}}} < \epsilon$$

where $\mathbf{e}^k$ is the error at the k^{th} iteration defined as $\mathbf{e}^k = \mathbf{x}^E - \mathbf{x}^k$, where $\mathbf{x}^E$ is the exact solution, and $\|\mathbf{e}\|_{\mathbf{A}} = \sqrt{e^T A\, e}$ is the A-norm of the error.

Proof: As in Eqn. (5.25), the Jacobian of the system at each Newton iteration is given by

$$\mathbf{J}_F\left(\mathbf{v}(t_{m+1})\right) = \frac{2}{h}\,\mathbf{C} + \frac{\partial \mathbf{f}\left(\mathbf{v}(t_{m+1}), \beta(t_{m+1})\right)}{\partial \mathbf{v}}.$$

Let $\mathbf{G} = \frac{\partial \mathbf{f}(\mathbf{v}(t_{m+1}), \beta(t_{m+1}))}{\partial \mathbf{v}}$ and let λ_m and λ_M be the magnitudes of respectively the smallest and largest eigenvalues of $\mathbf{G}$. Clearly then, as $\mathbf{C} = c\mathbf{I}$, the eigenvalues of $\mathbf{J}_F \in [\lambda_m + \frac{c}{h}, \lambda_M + \frac{c}{h}]$. Therefore the condition number of $\mathbf{J}_F$ is given by

$$\kappa(\mathbf{J}_F) = \frac{\lambda_M + \frac{c}{h}}{\lambda_m + \frac{c}{h}} = \frac{h\ \lambda_M + c}{h\ \lambda_m + c} \tag{5.26}$$

Since it is well known that [Trefethen88, Hesteness52]

$$\frac{\|\mathbf{e}^{k+1}\|_{\mathbf{A}}}{\|\mathbf{e}^{k}\|_{\mathbf{A}}} \leq 2\left(\frac{\sqrt{\kappa(\mathbf{J}_F)} - 1}{\sqrt{\kappa(\mathbf{J}_F)} + 1}\right).$$

Given an ϵ, the theorem will be proved if it can be shown that $\kappa(\mathbf{J}_F) < \frac{2+\epsilon}{2-\epsilon}$. It is sufficient to set

$$h < c\ \frac{\frac{2+\epsilon}{2-\epsilon} - 1}{\lambda_M - \lambda_m \frac{2+\epsilon}{2-\epsilon}}$$

∎

Note that in Theorem 5.1 the values of the capacitors to ground at each node are assumed to be the same. This condition can be relaxed if diagonal preconditioned Conjugate Gradient is used. We will not further study the matter, since it lends no insight for the test problems we study here.

5.3.2 Parallel Simulation of Vision Circuits on the Connection Machine

That the underlying structure of many of the analog signal processing circuits is a grid, and that the grid is made up of only a few different types of circuit elements, makes it possible to specialize the numerical algorithms to exploit the structure of the *Connection Machine*. The implementation hereby described was performed on a *CM–2*.

The most important characteristic of the CM that affects how well it will perform on certain problems is its SIMD (single instruction, multiple data) architecture. As we have known, this implies that all processors execute the same instruction simultaneously, although processors may be optionally inactive. Problems which can be solved well on the *Connection Machine* are those which exhibit a high degree of *data parallelism*. To make this statement more clear, by data parallelism we mean that

- One can map individual pieces of data to individual processors identically for all relevant processors,
- One can operate identically on the data with all the relevant processors.

We saw in chapter 4 that interprocessor communication on the *Connection Machine* is expensive (in terms of equivalent computation time) except for nearest-neighbor communication. So, a problem is better suited for the CM if, in addition to exhibiting data-parallelism, it is also local and regular. By this we mean that communication between processors should take place in a regular and localized fashion (e.g., nearest-neighbor on a 2-dimensional grid).

As we discussed in section 5.2, the general circuit simulation problem is not well suited to the *CM* . However simulation of the two-dimensional grid-based circuits discussed above is well suited to the CM. These circuits exhibit a high degree of data-parallelism and furthermore, when using a CG type algorithm, the linear system solution requires only nearest-neighbor communication between processors. To describe the details of the implementation more thoroughly, we present the core algorithm in algorithm 5.5.

We note that this methodology follows the direct methods approach except that it replaces the conjugate gradient algorithm for sparse Gaussian-elimination on the solution of the linear system of equations at each Newton iteration.

The data parallel paradigm compels us to use the CM for operations on the data, i.e., for device evaluation (within Newton's method) and the linear system solution. The main points to address are the circuit to processor mapping, the device evaluation process, and the CG iteration. The simulation program itself uses MIT's SIMLAB circuit simulator [SimlabUG,

Algorithm 5.5 (Circuit Simulation)

```
cm_simulate()
{
  t = 0 ;
  formulate the circuit equations ;
  while (t < Simulation_Time) {
    choose h ;
    repeat {
      use Newton's method to solve the nonlinear algebraic equations;
      solve the resulting linear algebraic system with CG;
    }
    until (converged) ;
    t = t + h ;
  }
}
```

SimlabPG] as a basis, and the parallel portions of the code were written in a CM dialect of the "C" programming language that is later translated to Paris, the *Connection Machine* assembly language. The data to processor mapping described is mostly due to Andrew Lumsdaine.

5.3.2.1 Data to Processor Mapping

As discussed in Chapter 4, the CM communication network is a 12-dimensional hypercube, but by properly enumerating the processors and by restricting communication to a subset of the total number of paths, one can embed any lower-dimensional grid into the CM. This type of embedding is referred to as a *geometry*. Since the grid-based circuits we have been discussing are so regular, there is a very natural mapping of them onto the CM processors.

In particular, we choose a two-dimensional geometry and map the circuit to the CM processors in such a way as to maintain data parallelism, regularity, and locality. We do this by dividing the circuit into identical cells, as shown in Fig. 5-15, and assigning to each processor the data associated with each cell. This would include the voltages at each circuit node in the cell, currents and charges at each node, etc. The use of the two-dimensional geometry allows us to perform all communications using the NEWS facilities instead of the generic router, therefore minimizing the time spent communicating.

Notice that in Fig. 5-15, the elements in each cell actually cross the cell boundaries. This sort of model would imply that one processor must be able to directly access data from the memory of another processor, which in fact cannot be done on the CM. Rather, one must

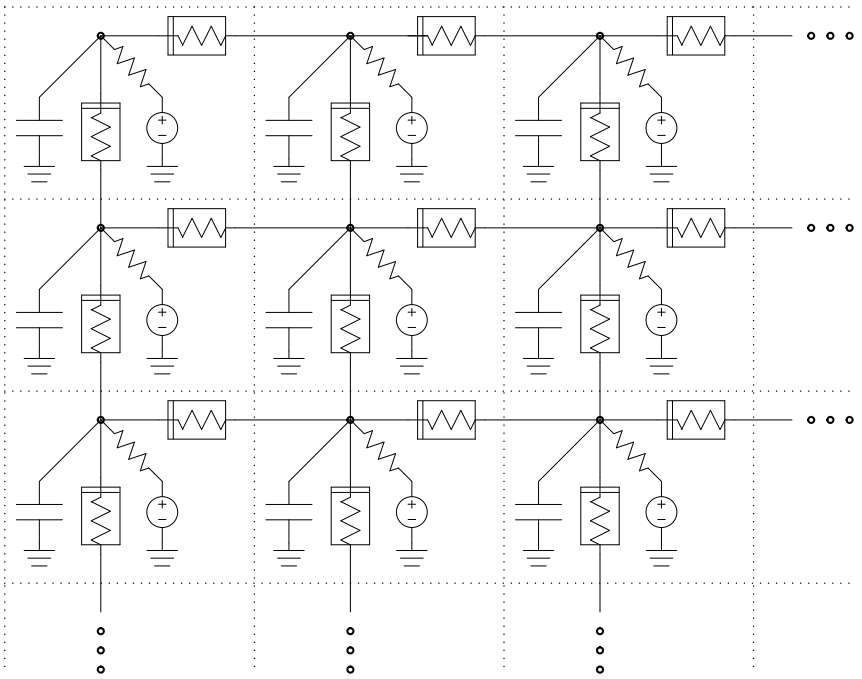

FIGURE 5-15: Division of the circuit into identical cells.

perform explicit communication steps. We can model this communication behavior from the circuit point of view. Within each cell, we make a copy of the node to the east and a copy of the node to the south, thereby allowing us to keep the cell devices completely within the cell. Such a structure is depicted in Fig. 5-16 where the capacitors to ground at each node have been omitted for simplicity. We then establish the convention that these copies are the "same" node as the node they were copied from. On the CM, this amounts to keeping copies of the node voltages, currents, and charges in neighboring processors and communicating them at appropriate times to maintain consistency. We discuss what these "appropriate times" are in the following discussion of device evaluation.

This particular mapping of the circuit data has some interesting implications for the simulation program. In particular, we can essentially pretend to simulate the simple circuit contained in each cell. However, by putting the data (i.e., node voltages, currents, and charges) for this simple circuit onto individual CM processors and enforcing certain consistencies in the data between neighboring processors, we actually simulate the original huge circuit grid.

Note that the cells on the east and south circuit boundaries are handled in a slightly different, but straightforward, manner. The differences will be pointed out briefly when convenient in the following discussion.

5.3.2.2 Device Evaluation

The device evaluation portion of the simulation is done in three stages:

1. Transfer node voltages west and north

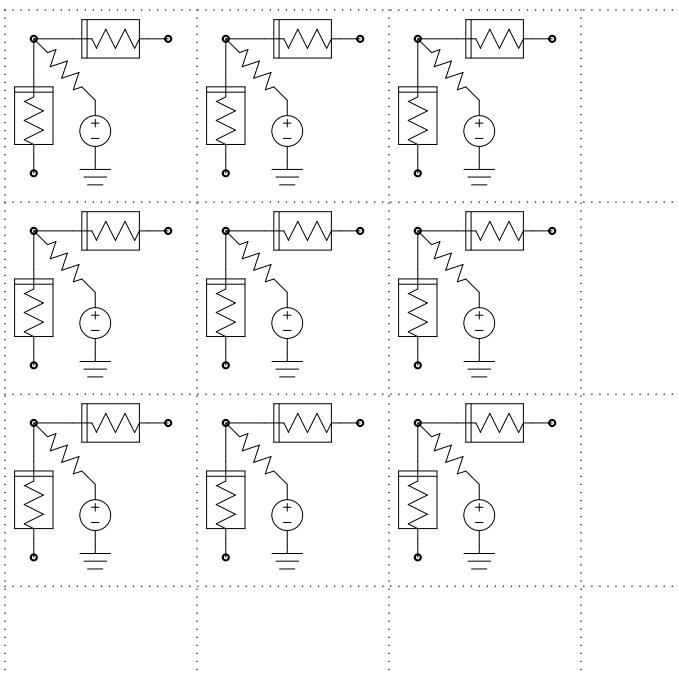

Figure 5-16: Separation of the cells by duplicating nodes (capacitors to ground omitted for simplicity).

2. Evaluate cell devices in parallel

3. Transfer element currents and node charges east and south

The first stage guarantees the consistency of neighboring node voltages and is accomplished by an explicit communication step. The second step corresponds to the actual device evaluation, whereby each individual processor evaluates the devices in the simple cell circuit. Consistency is ensured with a further communication step sending back the charges and currents computed for the inter-cell elements. Note that each processor has copies of its east and south neighbors and can compute the currents flowing into those nodes from devices in that cell. This intermediate step corresponds to local communication only. Clearly, more complex devices will need more computations to be performed which will improve the computation to communication ratio as well as the efficiency of the implementation. This immediately implies that more complex grids involving MOS grid elements will be simulated more efficiently than the resistive grids used as examples for this study.

The device evaluation process as described has a simple circuit interpretation as depicted in Fig. 5-17 and Fig. 5-18 where again the capacitors to ground at each node have been omitted for simplicity.

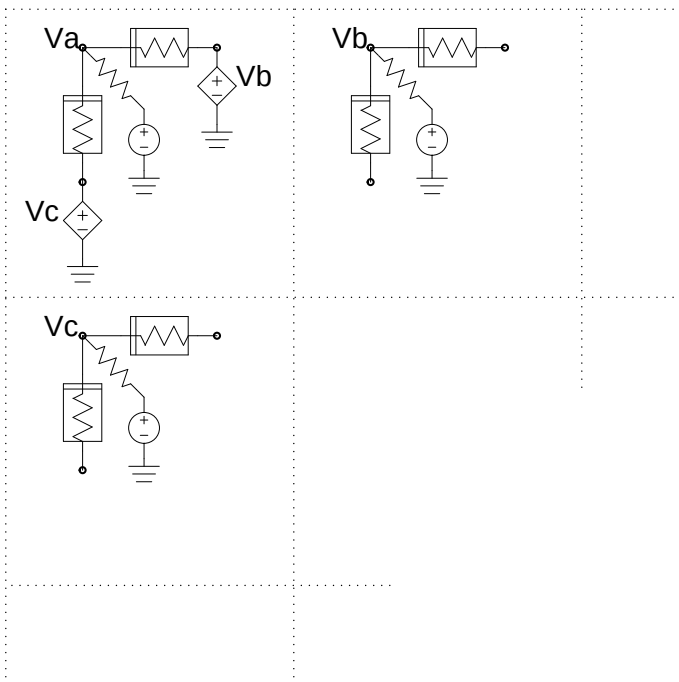

FIGURE 5-17: Circuit interpretation for maintaining node voltage consistency (capacitors to ground omitted for simplicity).

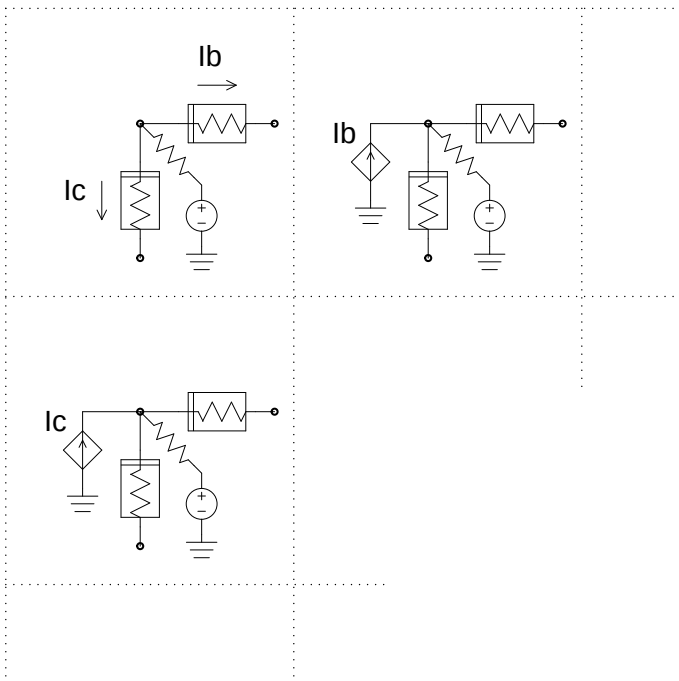

FIGURE 5-18: Circuit interpretation for maintaining node current consistency (capacitors to ground omitted for simplicity).

5.3.2.3 Conjugate Gradient Iteration

We remark that structurally the large circuit matrix is tridiagonal (from the connections along the horizontal lines) with an extra upper and lower diagonals (from the connections along the vertical lines. It is well known that this structure is disadvantageous for sparse gaussian elimination, but is very well suited for iterative algorithms using matrix-vector products, such as CG.

There are two parts of the CG iteration as shown in algorithm 5.4 which are handled in a unique fashion in our particular implementation:

1. The matrix-vector product, i.e., calculation of $\mathbf{w} = \mathbf{A}\,\mathbf{x}$, and
2. Vector inner product, i.e., calculation of $\langle \mathbf{r}, \mathbf{r} \rangle$ and $\langle \mathbf{p}, \mathbf{w} \rangle$.

The matrix-vector product is accomplished with a set of floating point operations and two communication steps. The operations involved in the matrix-vector product are essentially the same as those required for the device evaluation. The communication steps are still required for consistency, and the device evaluation step is now replaced by an in-place matrix-vector product where the local matrix corresponds to the linearized conductance matrix of the simple cell circuit. Again, more complex cells circuits will require more computation in the intermediate step of the matrix-vector product procedure, which again will improve the computation to communication ratio.

In Theorem 5.1 we proved that it is possible to arbitrarily improve the convergence rate of the Conjugate Gradient iteration during transient simulation. A similar result can also be found for DC simulation of linear grids as shown in Theorem 5.2. Conjugate Gradient DC simulation of nonlinear grids has also been found to have a problem size independent lower bound on the convergence rate, although that is not guaranteed.

CONJUGATE GRADIENT DC SIMULATION OF ANALOG LINEAR GRIDS

Theorem 5.2 *Let the Conjugate-Gradient algorithm be used to solve for the linear system of equations in Eqn. (5.23) generated at each Newton iteration. If the grid is formed with linear resistors, there is a resistor to ground at each grid node and the resistor values are within some range* $[r_m, r_M]$ *then there is a problem size independent lower bound on the rate of convergence of the Conjugate Gradient iteration.*

Proof: Since it is known that for the CG iteration

$$\frac{\|\mathbf{e}^{k+1}\|_{\mathbf{A}}}{\|\mathbf{e}^{k}\|_{\mathbf{A}}} \leq 2\left(\frac{\sqrt{\kappa(\mathbf{A})}-1}{\sqrt{\kappa(\mathbf{A})}+1}\right),$$

where again $\|\mathbf{e}\|_{\mathbf{A}}$ is the A-norm of the error, the theorem will be proved if it can be shown that there is a bound on the condition number of the matrix that is independent of the matrix

size. The existence of a resistor to ground at each node implies that the conductance matrix is strictly diagonal dominant, and the fact that the resistor values are in the range $[r_m, r_M]$, implies that the condition number is bounded by

$$\kappa(\mathbf{A}) = \rho(\mathbf{A})\,\rho(\mathbf{A}^{-1}) \leq \max_i(2a_{ii})\,\frac{1}{\min_i(a_{ii} - \sum_{i \neq j}|a_{ij}|)} \leq \frac{10\,r_M}{r_m},$$

which is independent of the problem size. ∎

5.3.3 Simulation Results

A set of experiments was devised to determine the suitability of the implementation to the simulation of the analog grids described. The set of experiments described in Table 5-6 is based on simple linear grids of resistor of different sizes and with different values of the resistor to ground (also known as the vertical resistor). In Table 5-6 g refers to the vertical conductance and r to the horizontal resistor in the grid. Changes in the value of g induce changes in the diagonal elements of the conductance matrix and therefore in the positioning of the eigenvalues. Increasing g^{-1} leads to an increasingly less diagonally dominant matrix and therefore worsens the conditioning of the matrix.

Experiment name	r (Ω)	g^{-1} (Ω)
L_{200}	1K	200
L_{1K}	1K	1K
L_{5K}	1K	5K
L_{25K}	1K	25K

Table 5-6: Linear Resistor Grid Experiments.

Table 5-7 contains the DC results obtained. The transient results are not of great interest since a DC solution is found previously and the circuit contains no time-dependent sources and were therefore omitted. For the *Connection Machine* experiments a 16K *CM–2* with double precision floating-point hardware was used and the VP ratio for each experiment is shown.

The results show, as expected, that the *CM* DC simulation time is approximately constant, independent of the size of the grid, as long as it fits on the physical machine. This can clearly be seen on the 64×64 and 128×128 in which constant time simulation is shown, and a VP ratio of 1 is used. If not enough processor are present, some penalty is paid which is close to the VP ratio for high VP ratios. In the 256×256 grid a VP ratio of 4 gives a penalty somewhere between 3 and 4. Also from the results we see that the various experiments cause the DC solution to have more or less difficulty, and the same changes show up in the *CM* results.

Examination of the timings for the larger grids reveal that for the 128×128 grid a speedup of around 170 can be expected with the *CM* implementation. Extrapolation of the 256×256 grid

Experiment	Grid Size	CPU Time (seconds)	
		Sequential	CM (VP ratio)
L_{200}	64×64	34.77	0.82 (1)
	128×128	169.5	0.96 (1)
	256×256	–	3.23 (4)
L_{1K}	64×64	64.68	1.53 (1)
	128×128	317.58	1.80 (1)
	256×256	–	6.05 (4)
L_{5K}	64×64	138.62	3.25 (1)
	128×128	638.0	3.77 (1)
	256×256	–	12.45 (4)
L_{25K}	64×64	275.9	6.44 (1)
	128×128	1265.45	6.83 (1)
	256×256	–	26.19 (4)

Table 5-7: DC solution results for the linear resistor grid experiments.

results indicates that speedups on the order of 920 can be obtained on a full 64K machine. Larger speedups can realistically be expected as the sequential version runs into memory problems for larger grids and the performance clearly deteriorates.

Table 5-8 describes the experiments that were run based on the grids containing the nonlinear elements described by the constitutive relation given in Eqn. (5.20). Some parameters were changed from experiment to experiment in order to better understand the results that can be expected in differing situations. In Table 5-8, g refers to the vertical conductance and and λ_l is a parameter in Eqn. (5.20). Changes in the vertical resistor to ground induce changes in the diagonal elements of the conductance matrix and therefore in the clustering of the eigenvalues. Increasing g^{-1} leads to an increasingly less diagonally dominant matrix and therefore worsens the conditioning of the matrix. Also, changes in λ_l cause the nonlinearity to become more or less visible in the circuit.

Experiment name	λ_l (Ω^{-1})	g^{-1} (Ω)
L_{1e-4}	0.0001	33K
L_{2e-5}	0.00002	33K
L_{5e-6}	0.000005	33K
L_{10K}	0.00002	10K

Table 5-8: Nonlinear resistor grid experiments.

Table 5-9 contains the DC and transient results obtained. In this case the transient results are the more interesting ones since a continuation is made on the nonlinear horizontal elements that changes the conditioning of the matrix with time. For the *Connection Machine* experi-

ments a 16K *CM–2*, with double precision floating-point hardware was used The VP ratio on each experiment is shown.

Experiment	Grid Size	CPU Time (seconds)			
		Sequential		CM (VP ratio)	
		DC	Transient	DC	Transient
L_{1e-4}	64×64	229.48	369.13	6.05 (1)	9.26
	128×128	984.91	1893.67	6.01 (1)	12.31
	256×256	–	–	22.44 (4)	388.68
L_{2e-5}	64×64	232.85	4598.9	5.98 (1)	120.28
	128×128	1000.4	45633.2	6.11 (1)	302.25
	256×256	–	–	22.21 (4)	2946.09
L_{5e-6}	64×64	223.8	364.57	5.93 (1)	9.10
	128×128	999.78	1926.68	6.27 (1)	11.17
	256×256	–	–	22.26 (4)	383.62
L_{10K}	64×64	142.45	7218.63	3.35 (1)	172.76
	128×128	–	–	5.86 (1)	10.52
	256×256	–	–	22.7 (4)	376.35

Table 5-9: Results of the nonlinear resistor grid experiments.

The results shown in Table 5-9 again show that, besides a penalty factor paid when using virtual processing, simulation in constant time is possible irrespective of the circuit size. Unfortunately, simulation of the larger grids on sequential machines is very expensive and we must therefore resort to some extrapolations to capture the speedups that can be expected. Depending on the experiment used, different speedups are obtained. For the 128×128 grids a speedup around 160 can be expected for DC simulation, while only a factor of 150 is obtained for transient simulation. Extrapolation of the results for a 256×256 grid indicates that a speedup of 750 can be expected on DC simulation, while extrapolation of the transient results on experiment L_{2e-5} indicates speedups around 650 can be obtained. The slight discrepancy between DC and transient results is understandable as the transient simulation requires more front-end interaction and therefore more serialization and synchronization between the front-end and the *CM*. The DC simulation, on the other hand, is mostly CG and therefore will perform better overall since less front-end overhead is involved.

Overall, the results must be considered very encouraging. If one considers that a simulation of a 128×128 grid takes over 10 hours on a conventional workstation and the *CM* implementation allows that same simulation to be performed in the span of a few minutes, we can see that simulating large analog grids on a machine like the *CM* is indeed worthwhile. Furthermore, the simulation of larger grids is an almost unthinkable task on a sequential computer due not only to unacceptable simulation time but also to memory limitations.

There is whole world of possibilities opened by this implementation. We strongly emphasize

that the experiments presented in this thesis are very simple grids of simple and idealized elements. The simulator described can nevertheless simulate actual VLSI circuitry. This circuitry, such as the grids envisioned by the MIT vision chip project, contain cells of much a more intricate nature that makes them impossible to simulate on conventional computers. However, the *CM* implementation will be able to simulate them within a reasonable time span, which opens the possibility of actually testing this type of circuits before fabrication.

6

Conclusions

This thesis is concerned with studying the suitability of the recently available massively parallel computers as platforms for circuit simulation. The class of problems that can be efficiently solved with this type of architecture must satisfy a certain number of features that allows them to take advantage of the massive number of computational units available. Numerical algorithms with a high-degree of parallelism were investigated and implementation results reported. It was shown that regularity and locality are essential characteristics that circuits must have for efficient simulation to be performed on massively parallel computers.

6.1 Summary of Results

In this thesis we have considered the computational tasks involved in circuit simulation and determined how they can be parallelized. Evaluation of the model elements is easily parallelized provided an efficient parallel model is implemented. Linear systems solution on the other hand has proved to be a more difficult task.

To avoid solving the linear systems of equations, we studied a class of first order explicit integration formulas, namely explicit exponential fitting integration formulas. Their stability properties have been properly studied and a unifying relation was established between several of the algorithms used in the past. These formulas present lower complexity and a high-degree of parallelism but are not more accurate than backward-Euler for large timesteps and nondiagonal problems. Some of the formulas studied, such as the CINNAMON algorithm, have been shown to be stable but suffer from the fact that as the timesteps are increased they are unable to follow the dynamics of the system, instead getting "stuck".

Ordering the equations was shown to improve the accuracy of all methods. In general however, the increase in accuracy obtained from ordering the equations is accompanied by a decrease in exploitable parallelism, since ordering implies serialization of the computations. Furthermore, it was shown that efficient ordering heuristics require substantial effort. For these reasons, these methods were not further investigated and instead more accurate implicit

formulas were pursued.

Multiple timepoint algorithms were studied in order to try to increase parallelism which would insure that all the processors are kept busy. Although these methods have the attractive advantage that they increase the parallelism exploitable in the simulation problem, they are more demanding in terms of communication and there seems to be no simple and effective way to place the timepoints "a priori". These methods were therefore considered not suitable for implementation on massively parallel computers and their study abandoned.

Band-Relaxation was also studied and an implementation on the *Connection Machine* described. Band-Relaxation is aimed at improving the reliability of Gauss-Jacobi relaxation while at the same time keeping a high-degree of parallelism. The results obtained show that although efficient nonlinear band-relaxation for moderately to large circuits is possible in a parallel architecture, taking into account the amount of raw computational devices, the speedups obtained were below expectations. It was argued that the irregular topology of circuits does not make them a good candidate for implementation on massively parallel machines, where communications performed in an irregular fashion is very expensive.

Restricting the class of circuits to those which exhibit inherent structural regularity was shown to help improve circuit simulation performance. In particular, an implementation on the *Connection Machine* was described that is aimed at simulating analog grid circuits. These circuits perform early vision tasks, such as smoothing and segmentation, and in general cannot be simulated with traditional programs on traditional computers, due to their large size. An innovative approach for implementing an efficient circuit simulator for this type of circuit on a massively parallel machine was described. This simulator is based on a two-dimensional layout of data and strictly follows the SIMD paradigm to allow for more efficient device evaluation. In addition, the conjugate gradient algorithm is used to solve for the linear system at each Newton iteration. Speedups of two or three orders of magnitude on certain types of grids were reported and more important it was shown that simulation of these grids is possible in constant time independent of the grid size.

6.2 Future Work

The band-relaxation implementation described was shown to be more efficient than a similar Gauss-Jacobi relaxation algorithm. Nevertheless the irregular circuit topology requires a irregular and therefore costly communication pattern which decreases efficiency. Furthermore, the solution of the banded system does not present enough parallelism for a massively parallel algorithm. Blocking also improves the convergence of Gauss-Jacobi relaxation, may provide more parallellism than band-relaxation, and the two should therefore be compared. Block-based relaxation algorithms should also be studied and compared against band-relaxation to determine whether a more efficient way of grouping sets of elements is possible that actually

blocks the circuit equations therefore making it simpler to solve them in parallel. Nevertheless we note that generic circuit simulation does not scale well with circuit size due to the irregularity which makes it more difficult to exploit the numerous processing elements available on a massively parallel architecture. it is our belief that the speedups that are possible to obtain on a machine such as the *Connection Machine* are within the tenths

The performance of the analog grid simulator on the *Connection Machine* allows for significantly large and intricate circuitry to be accurately simulated before fabrication. Circuits containing more complex individual cells composed of semi-conductor elements should be considered in order to determine how efficiently they can be simulated. With the addition of more robust conjugate gradient-like techniques such as preconditioning, the convergence rate can be greatly increased. These algorithms nevertheless require some communication to be performed before the matrix-vector product, and some investigation is necessary to determine whether they will pay off in the long run.

A

The Simlab Simulation Environment

SIMLAB is a circuit simulation environment consisting of a flexible, user-friendly front-end operating in conjunction with a sophisticated and versatile simulation engine. The program is specifically designed to be used as an educational tool and as a research platform [SimlabUG, SimlabPG].

Since its inception, SIMLAB has been used for two distinct purposes:

1. circuit simulation and
2. algorithm development for circuit simulation.

However, SIMLAB is primarily intended to be a research platform for circuit simulation algorithms. As such, one of its major features is its modularity, which allows customization of the base program through a simple functional interface. There are three main areas of the program for which customization is easily facilitated:

- Adding a new device type
- Adding a new numerical algorithm
- Adding a new interactive command

Adding a new device type in SIMLAB amounts to writing a single module containing the necessary parsing structures to handle the device module fields and also some evaluation functions. This modularity makes SIMLAB an easily expandable environment. Devices modules have already been incorporated into SIMLAB through this via.

Adding a new numerical algorithm can be done in several steps. SIMLAB has notions of discretization formulas, linear and nonlinear solvers, and specific algorithms for each of these tasks can be specified separately. If the user wants to try, say, a numerical integration formula

of his choice, that simply implies adding a module that can handle the numerical integration formula. Since the discretization of the ordinary differential equations and the solution of the resulting nonlinear algebraic equations can be set separately, a numerical integration module simply has to appropriately handle values of previous time iterations to form the right-hand side in Newton's multidimensional method and the calculations of truncation errors. The same is true for all other aspects of the simulation process. The program code is in fact highly modular, so that researchers can easily construct and test algorithms by inserting them into the existing SIMLAB framework. On the top level, SIMLAB has the notion of environment, such as sequential or CM, which allows versions for intrinsically different machines to be built on top of it, sharing common functionality.

New commands can also be added to work upon the program structures. This flexibility makes it extremely appealing to customize SIMLAB for different applications. During the time that this research was conducted, SIMLAB has formed the core of special-purpose simulation programs, such as a switched capacitor filter simulator and a simulator for vision circuits.

As a general purpose circuit circuit simulator, SIMLAB can be operated in either batch or interactive mode, giving the user the possibility to separately specify algorithms for six aspects of the simulation:

- the solution type for the system of ordinary differential equation (e.g., pointwise solution, waveform Newton, waveform relaxation)
- the time integration formula for the system of ordinary differential equations (e.g., backward-Euler, trapezoidal, second-order Gear)
- transient nonlinear algebraic system solution (e.g., multidimensional Newton's method, nonlinear relaxation),
- transient linear system solution (e.g., sparse Gaussian elimination, Gauss-Jacobi relaxation, band relaxation, conjugate gradient),
- DC nonlinear algebraic system solution (e.g., multidimensional Newton's method, nonlinear relaxation), and
- DC linear system solution (e.g., sparse Gaussian elimination, Gauss-Jacobi relaxation, band relaxation, conjugate gradient).

Although SIMLAB provides default values offering the desired functionalities, being able to insert new algorithms into the simulator and then interactively specify their use, provides the user with the possibility of testing his ideas and algorithms on a real problem without the burden of having to write a complete simulator from scratch.

The use of SIMLAB as an educational tool and as a research tool has proved invaluable during this research. All the algorithms mentioned in this thesis and their parallel implementations were performed and investigated on the SIMLAB framework.

Bibliography

[Bauer88] R. Bauer, J. Fang, A. Ng, and R. Brayton, "XPSim: A MOS VLSI Simulator", in *Proceedings International Conference on Computer-Aided Design*, (Santa Clara), November 1988.

[Chawla75] B. R. Chawla, H. K. Gummel, and P. Kozah, "MOTIS - an MOS Timing Simulator", *IEEE Transactions on Circuits and Systems*, vol. CAS-22, pp. 901–909, December 1975.

[Chen84] C. F. Chen and P. Subramanyam, "The Second Generation MOTIS Timing Simulator - An Efficient and Accurate Approach for General MOS Circuit", in *Proceedings International Symposium on Circuits and Systems*, (Montreal, Canada), May 1984.

[Chua87] L. O. Chua, C. A. Desoer, and E. S. Kuh, *Linear and Nonlinear Circuits*. Circuits and Systems, New York: McGraw-Hill, 1987.

[CM2] Thinking Machines Corporation, Cambridge, Massachusetts, USA, *Connection Machine Model CM-2 Technical Summary*, May 1989. Technical Report-General TR 89-1.

[CMDataPar] Thinking Machines Corporation, Cambridge, Massachusetts, USA, *Introduction to Data Level Parallelism*, April 1986. Technical Report-General TR 86-14.

[Dahlquist74] G. Dahlquist and Å. Björck, *Numerical Methods*. Automatic Computation, Englewood Cliffs, New Jersey: Prentice-Hall, 1974.

[Desoer69] C. A. Desoer and E. S. Kuh, *Basic Circuit Theory*. New York: McGraw-Hill, 1987.

[Deutsch85] J. T. Deutsch and A. R. Newton, "A Multiprocessor Implementation of Relaxation-Based Electrical Circuit Simulation", in *Proceedings* 21^{nd} *ACM/IEEE Design Automation Conference*, 1985.

[Dongarra87] J. Dongarra and S. L. Johnsson, "Solving Banded Systems on a Parallel Processor", *Parallel Computing*, vol. 5, no. 1&2, pp. 219–246, 1987.

[Duff86] I. S. Duff, A. M. Erisman, and J. K. Reid, *Direct Methods for Sparse Matrices*. Oxford: Clarendon Press, 1986.

[Flynn66] M. J. Flynn, "Very High Speed Computing Systems", in *Proceedings IEEE*, vol. 54, pp. 1901–1909, 1966.

[Gallivan] K. A. Gallivan, P. Koss, S. Lo, and R. A. Saleh, "A Comparison of Parallel Relaxation-Based Circuit Simulation Techniques", Tech. Rep., Center for Supercomputing Research, U. Illinois-Urbana, Urbana, Illinois, .

[George81] A. George and J. W.-H. Liu, *omputer Solution of Large Sparse Positive Definite Systems*. Automatic Computation Series, Englewood Cliffs, New Jersey: Prentice-Hall, 1981.

[Golub83] G. H. Golub and C. F. V. Loan, *Matrix Computations*. Baltimore, Maryland: The John Hopkins University Press, 1983.

[Hancock86] J. M. Hancock and S. DasGupta, "Tutorial on Parallel-Processing for Design Automation Applications", in *Proceedings* 23^{rd} *ACM/IEEE Design Automation Conference*, (Las Vegas, Nevada), July 1986.

[Hesteneс52] M. R. Hestenes and E. Stiefel, "Methods of Conjugate Gradients for Solving Linear Systems", *Journal of Research of the National Bureau of Standards*, vol. 49, pp. 409–436, December 1952.

[Hillis85] W. D. Hillis, *The Connection Machine*. Cambridge, Massachusetts: MIT Press, 1985.

[Ho75] C. W. Ho, A. E. Ruehli, and P. A. Brennan, "The Modified Nodal Approach to Network Analysis", *IEEE Transactions on Circuits and Systems*, vol. CAS-22, pp. 504–509, June 1975.

[Hockney88] R. W. Hockney and C. R. Jesshope, *Parallel Computers 2*. Adam Hilger, 1988.

[Horn86] B. K. P. Horn, *Robot Vision*. Cambridge, MA: MIT Press/McGraw-Hill, 1986.

[Horn88] B. K. P. Horn, "Parallel Networks for Machine Vision", Tech. Rep. AI Memo 1071, Artificial Inteligence Laboratory, Massachusetts Institute of Technology, Cambridge, MA, August 1988. Unpublished.

[Hutchinson88] J. Hutchinson, C. Koch, J. Luo, and C. Mead, "Computing Motion Using Analog and Binary Resistive Networks", *IEEE Computer*, pp. 52–63, March 1988.

[Hwang84] K. Hwang and F. A. Briggs, *Computer Architecture and Parallel Processing*. New York: McGraw-Hill, 1984.

[Jacob86] G. Jacob, A. R. newton, and D. Pederson, "An Empirical Analysis of the Performance of a Multiprocessor-based Circuit Simulator", in *Proceedings* 23^{rd} *Design Automation Conference*, (Las Vegas, Nevada), pp. 588–593, June 1986.

[Kantorovich64] L. V. Kantorovich and G. P. Akilov, *Functional Analysis in Normed Spaces*. Oxford: Pergammon Press, 1964.

[Kim88] Y. H. Kim, S. H. Hwang, and A. R. Newton, "Electrical-Logic Simulation and its Applications", *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 8, pp. 8–22, January 1988.

[Koch85] C. Koch, J. Marroquin, and A. Yuille, "Analog 'Neuronal' Networks in Early Vision", Tech. Rep. AI Memo 751, Artificial Inteligence Laboratory, Massachusetts Institute of Technology, Cambridge, MA, June 1985.

[Kundert86] K. S. Kundert, "Sparse Matrix Techniques", in *Circuit Analysis, Simulation and Design* (A. E. Ruehli, ed.), pp. 281–324, North-Holland, 1986.

[Leighton88] F. T. Leighton, "An Introduction to the Theory of Networks, Parallel Computation and VLSI Design". Lecture notes, Massachusetts Institute of Technology. Unpublished, 1988.

[Lelarasmee82] E. Lelarasmee, A. E. Ruehli, and A. L. Sangiovanni-Vincentelli, "The Waveform Relaxation Method for Time Domain Analysis of Large Scale Integrated Circuits", *IEEE Transactions on CAD of IC and Systems*, vol. 1, pp. 131–145, July 1982.

[Levit88] C. Levit, "Grid Communication on the Connection Machine: Analysis, Performance and Improvements", in *Proceedings of the Conference on Scientific Applications on the CM* (H. D. Simon, ed.), pp. 316–332, World Scientific, Singapore, September 1988.

[Lumsdaine88] A. Lumsdaine, J. White, D. Webber, and A. Sangiovanni-Vincentelli, "A Band Relaxation Algorithm for Reliable and Parallelizable Circuit Simulation", in *Proceedings International Conference on Computer Aided Design*, (Santa Clara, California), November 1988.

[Lumsdaine89] A. Lumsdaine, J. Wyatt, and I. Elfadel, "Parallel Distributed Networks for Image Smoothing and Segmentation in Analog VLSI", in *Proceedings 28th IEEE Conference on Decision and Control*, pp. 272–279, December 1989.

[Lumsdaine90] A. Lumsdaine, J. Wyatt, and I. Elfadel, "Nonlinear Analog Networks for Image Smoothing and Segmentation", in *International Symposium on Circuits and Systems*, (New Orleans, Louisiana), pp. 987–991, May 1990.

[Marr82] D. Marr, *Vision*. San Francisco, CA: W.H. Freeman & Co., 1982.

[Mcbryan88] O. A. McBryan, "Connection Machine Application Performance", in *Proceedings of the Conference on Scientific Applications on the CM* (H. D. Simon, ed.), pp. 94–115, World Scientific, Singapore, September 1988.

[Mead88] C. Mead, *Analog VLSI and Neural Systems*. Reading, MA: Addison-Wesley, 1980.

[Myers88] D. W. Myers and G. B. A. III, "Benchmarking and Performance Analysis on the CM-2", in *Proceedings of the Conference on Scientific Applications on the CM* (H. D. Simon, ed.), pp. 333–349, World Scientific, Singapore, September 1988.

[Nagel75] L. W. Nagel, "SPICE2: A Computer Program to Simulate Semiconductor Circuits", Tech. Rep. ERL M520, Electronics Research Laboratory Report, University of Califórnia, Berkeley, Berkeley, Califórnia, May 1975.

[Neto89] H. Neto and L. Vidigal, "A Multirate Algorithm for Fast Circuit Simulation", in *Proceedings of the European Conference in Circuit Theory and Design*, (Brighton, England), September 1989.

[Neto90] H. Neto, L. M. Silveira, J. K. White, and L. M. Vidigal, "On Exponential Fitting for Circuit Simulation", in *International Symposium on Circuits and Systems*, (New Orleans, Louisiana), pp. 514–518, May 1990.

[Newton83] A. R. Newton and A. L. Sangiovanni-Vincentelli, "Relaxation - Based Circuit Simulation", *IEEE Transactions on Electron Devices*, vol. ED-30, pp. 1184–1207, September 1983.

[Ortega70] J. M. Ortega and W. C. Rheinbolt, *Iterative Solution of Nonlinear Equations in Several Variables*. Computer Science and Applied Mathematics, New York: Academic Press, 1970.

[Poggio84] T. Poggio and C. Koch, "An Analog Model of Computation for the Ill-Posed Problems of Early Vision", Tech. Rep. AI Memo 783, Artificial Inteligence Laboratory, Massachusetts Institute of Technology, Cambridge, MA, May 1984.

[Poggio85] T. Poggio, V. Torre, and C. Koch, "Computational Vision and Regularization Theory", *Nature*, vol. 317, pp. 314–319, 1985.

[Saleh83] R. A. Saleh, J. E. Kleckner, and A. R. Newton, "Iterated Timing Analysis and SPLICE1", in *Proceedings International Conference on Computer-Aided Design*, (Santa Clara, California), September 1983.

[Saleh87] R. A. Saleh and A. R. Newton, "An Event-Driven Relaxation-Based Multirate Integration Scheme for Circuit Simulation", in *Proceedings International Symposium on Circuits and Systems*, (Philadelphia, Pennsylvannia), May 1987.

[Sandberg74] I. W. Sandberg, "A nonnegativity-preservation property associated with certain systems of nonlinear differential equations", in *Proceedings International Conference on Systems, Man, and Cybernetics*, 1974.

[Sarkani74] E. Sarkani and W. Liniger, "Exponential Fitting of Matricial Multistep Methods for Ordinary Differential Equations", *Mathematics of Computation*, vol. 28, pp. 1035–1052, October 1974.

[Seitz85] C. L. Seitz, "The Cosmic Cube", *Communications of the ACM*, January 1985.

[Silveira88] L. M. Silveira, "A Parallel Band-Matrix Solver for a Circuit Simulation Problem". Internal Report, Massachusetts Institute of Technology and Thinking Machines Corporation. Unpublished, December 1988.

[SimlabPG] A. Lumsdaine, M. Silveira, and J. White, "SIMLAB Programmer's Guide". Research Laboratory of Electronics, Massachusetts Institute of Technology. Unpublished, 1989.

[SimlabUG] A. Lumsdaine, M. Silveira, and J. White, "SIMLAB Users' Guide". Research Laboratory of Electronics, Massachusetts Institute of Technology, 1989.

[Smart87] D. W. Smart and T. N. Trick, "Increasing Parallelism in Multiprocessor Waveform Relaxation", in *Proceedings International Conference on Computer Aided Design*, 1987.

[Terzopoulos84] D. Terzopoulos, "Multigrid Relaxation Methods and the Analysis of Lightness, Shading, and Flow", Tech. Rep. AI Memo 803, Artificial Inteligence Laboratory, Massachusetts Institute of Technology, Cambridge, MA, October 1984.

[Trefethen88] L. N. Trefethen, "Approximation Theory and Numerical Linear ALgebra". Unpublished, to appear in *Approximation Theory and Numerical Linear Algebra*, eds J. C. Mason and M. G. Cox, 1988.

[Trefethen89] L. N. Trefethen, "Finite-Difference and Spectral Methods". Lecture notes, Massachusetts Institute of Technology. Unpublished, 1989.

[Varga62] R. S. Varga, *Matrix Iterative Analysis*. Automatic Computation Series, Englewood Cliffs, New Jersey: Prentice-Hall Inc, 1962.

[Vidigal86] L. Vidigal, S. Nassif, and S. Director, "CINNAMON: Coupled INtegration and Nodal Analysis of MOs Networks", in *Proceedings 23^{rd} Design Automation Conference*, (Las Vegas, Nevada), June 1986.

[Vlach83] J. Vlach and K. Singhal, *Computer Methods for Circuit Analysis and Design*. Berkshire, England: Van Nostrand Reinhold, 1983.

[Wang81] H. H. Wang, "A Parallel Method for Tridiagonal Equations", *ACM Transactions on Mathematical Software*, vol. 7, pp. 170–183, 1981.

[Webber87] D. M. Webber and A. Sangiovanni-Vincentelli, "Circuit Simulation on the Connection Machine", in *Proceedings* 24^{th} *ACM/IEEE Design Automation Conference*, June 1987.

[Weeks73] W. T. Weeks, A. J. Jimenez, G. W. Mahoney, D. Mehta, H. Quasemzadeh, and T. R. Scott, "Algorithms for ASTAP - A Network Analysis Program", *IEEE Transactions on Circuit Theory*, pp. 628–634, November 1973.

[White85] J. White, R. Saleh, A. Sangiovanni-Vincentelli, and A. Newton, "Accelerating Relaxation Algorithms for Circuit Simulation using Waveform Newton, Iterative step size Refinement and Parallel Techniques", in *Proceedings International Conference on Computer Aided Design*, pp. 5–7, November 1985.

[White86] J. K. White and A. Sangiovanni-Vincentelli, *Relaxation Techniques for the Simulation of VLSI Circuits*. Engineering and Computer Science Series, Norwell, Massachusetts: Kluwer Academic Publishers, 1986.

[Wiley87] P. Wiley, "A Parallel Architecture comes of age at last", *IEEE Spectrum*, June 1987.

[Wyatt88] J. L. W. Jr and et all, "Smart Vision Sensors: Analog VLSI Systems for Integrated Image Acquisistion and Early Vision Processing". Massachusetts Institute of Technology. Unpublished, 1988.