July 1988

UILU-ENG-88-2237
DAC-12

COORDINATED SCIENCE LABORATORY
College of Engineering

PARALLEL PROCESSING TECHNIQUES FOR THE SIMULATION OF MOS VLSI CIRCUITS USING WAVEFORM RELAXATION

David William Smart

UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION Unclassified	1b. RESTRICTIVE MARKINGS None
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution unlimited
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S) UILU-ENG-88-2237 (DAC-12)	5. MONITORING ORGANIZATION REPORT NUMBER(S)
6a. NAME OF PERFORMING ORGANIZATION Coordinated Science Lab University of Illinois — 6b. OFFICE SYMBOL (If applicable) N/A	7a. NAME OF MONITORING ORGANIZATION Semiconductor Research Corporation; Joint Services Electronics Program; Sandia National Laboratory
6c. ADDRESS (City, State, and ZIP Code) 1101 W. Springfield Avenue Urbana, IL 61801	7b. ADDRESS (City, State, and ZIP Code) SRC: Research Triangle Park, NC 27709 JSEP: Arlington, VA 22217 Sandia: Albuquerque, NM
8a. NAME OF FUNDING/SPONSORING ORGANIZATION SRC, JSEP, SANDIA — 8b. OFFICE SYMBOL (If applicable)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER SRC: 87-DP-109; JSEP: N00014-84-C-0149; SANDIA: 02-8522
8c. ADDRESS (City, State, and ZIP Code) see block 7b	10. SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO. / WORK UNIT ACCESSION NO.

11. TITLE (Include Security Classification)
PARALLEL PROCESSING TECHNIQUES FOR THE SIMULATION OF MOS VLSI CIRCUITS USING WAVEFORM RELAXATION

12. PERSONAL AUTHOR(S)
Smart, David William

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15. PAGE COUNT
Technical	FROM Aug -85 TO June -88	1988 July	145

16. SUPPLEMENTARY NOTATION

17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	parallel processing, circuit simulation, waveform relaxation

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

Waveform relaxation algorithms for the simulation of MOS circuits exhibit natural parallelism, arising from the intrinsic partitioning of the circuit into subcircuits which are solved separately during the iterative solution process. Investigated in this thesis is the extent to which the overall run time of a simulation can be reduced by utilizing the natural parallelism of waveform relaxation on parallel processors. Four parallel waveform relaxation algorithms are considered, based on the Gauss-Seidel and

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT ☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT. ☐ DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION Unclassified
22a. NAME OF RESPONSIBLE INDIVIDUAL	22b. TELEPHONE (Include Area Code) — 22c. OFFICE SYMBOL

DD FORM 1473, 84 MAR — 83 APR edition may be used until exhausted. All other editions are obsolete.

SECURITY CLASSIFICATION OF THIS PAGE
UNCLASSIFIED

PARALLEL PROCESSING TECHNIQUES FOR THE SIMULATION OF MOS VLSI CIRCUITS USING WAVEFORM RELAXATION

BY

DAVID WILLIAM SMART

B.S., University of Illinois, 1976
M.S., University of Illinois, 1977

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Electrical Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 1988

Urbana, Illinois

ABSTRACT

Waveform relaxation algorithms for the simulation of MOS circuits exhibit natural parallelism, arising from the intrinsic partitioning of the circuit into subcircuits which are solved separately during the iterative solution process. Investigated in this thesis is the extent to which the overall run time of a simulation can be reduced by utilizing the natural parallelism of waveform relaxation on parallel processors. Four parallel waveform relaxation algorithms are considered, based on the Gauss-Seidel and Gauss-Jacobi relaxation methods, in which parallelism is exploited at the individual time point level or at the time window level. The algorithm with the fastest run time depends on the characteristics of the circuit being simulated and on the number of processors used. The Gauss-Jacobi method with time point pipelining is introduced as a highly parallel algorithm which can outperform the other algorithms when the number of processors is large.

A theorem is presented comparing the Gauss-Seidel and Gauss-Jacobi methods, as applied to the solution of a set of linear algebraic equations of the type which occur at each time point in the simulation of MOS circuits. Gauss-Jacobi is shown to be asymptotically faster than Gauss-Seidel when the number of processors is sufficiently large.

Simplified speedup estimates are used in a presimulation selection procedure which selects the fastest of the parallel waveform relaxation algorithms prior to performing the simulation of a given circuit on a given number of processors. More accurate estimates of the potential parallel processing speedups, neglecting overhead, are produced by the PARASITE parallel simulation time estimator, which uses CPU time measurements from a uniprocessor simulation to estimate the parallel run time on any number of processors. PARASITE estimates indicate that speedups of about one order of magnitude are possible for 1000-transistor circuits on 32 processors, where the speedup is measured with respect to Gauss-Seidel waveform relaxation on a single processor.

The parallel waveform relaxation algorithms have been implemented in programs which run on an 8-processor Alliant FX/8 multiprocessor. Speedups within 11% to 21% of the PARASITE estimates are achieved.

ACKNOWLEDGEMENTS

I would like to express my appreciation to my thesis advisor, Professor Timothy N. Trick, for his consistent support, his perpetual positive attitude, and his valuable guidance. I would also like to thank the other members of my dissertation committee: Professors Ibrahim Hajj, Vasant Rao, and Ahmed Sameh. In addition to these individuals, Professor Res Saleh, and other faculty members and students in the circuits group of the Coordinated Science Laboratory have provided interesting insights which have broadened my perspectives on VLSI circuits, simulation, and related topics.

The influence of Jacob White on this research has been multifaceted and is greatly appreciated. First of all, his doctoral research at the University of California at Berkeley, including the development of RELAX2.3, has served as a starting point for the work presented in this thesis. When I showed him some of my early results, his first reaction was, "Where are the theorems?" This question led to the development of the material in Chapter 3, to which Jacob contributed some of the key ideas.

I would like to thank Professors David Kuck and Ahmed Sameh for their cooperation in providing time on the Alliant FX/8 computers at the Center for Supercomputing Research and Development at the University of Illinois. I would also like to thank Gung-Chung Yang, Dan Sorensen, Kyle Gallivan, and Sy-Shin Lo at the Center for their helpfulness.

This research was supported by Sandia National Laboratory, the Joint Services Electronics Program, and the Semiconductor Research Corporation.

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

Circuit simulation plays a key role in verifying that very large scale integrated (VLSI) circuit designs are correct and meet performance specifications prior to fabricating the circuits [Nag75, Wee73]. The amount of computer time required to simulate a significant portion of a large circuit can be prohibitive when detailed models of the circuit elements are used, and the differential equations are integrated to produce accurate voltage waveforms as a function of time [Whi86]. Simulators that use models based on higher levels of abstraction, such as switch and logic level simulators [Bry84, Rao85, Szy72, Haj83], can achieve run times which are orders of magnitude faster, but the results are not as precise. As integrated circuit technology advances, the number of circuit elements increases at the same time that feature sizes are reduced and parasitic effects become more important in determining circuit performance. Consequently, as the need for faster simulation becomes more important, the ability to satisfy this need through less precise simulation techniques diminishes [Kan85].

The need for faster, precise circuit simulation that is motivated by advances in integrated circuit technology can be addressed by employing the fruits of these technological advances in the computers which are used to simulate circuits. Multiprocessor computers, comprised of several processors which can work together to solve a single problem, are able to provide a large amount of total computing power at a reasonable cost [Kuc86]. To successfully use the parallelism in the computer hardware to reduce the overall run time of the simulation program, the set of computations must be partitioned into subsets which can be executed currently on different processors, while preserving the integrity of the final solution.

Early attempts to exploit parallelism to reduce the time required to perform circuit simulations utilized vector processors, which achieve performance gains from parallelism when all the elements of a vector are processed identically. The vector elements are pipelined through the processor hardware such that different elements are processed by different stages of the hardware simultaneously [Hwa84]. The hardware parallelism is limited by the number of stages in the pipeline. The effective use of vector processing in standard direct method circuit simulation algorithms [Nag75, Wee73, Yan80] is hampered by the sparse, irregular interconnection structures of circuits, which result in highly sparse unstructured matrices that are not efficiently processed by vector operations. When circuit elements are evaluated, elements described by identical equations can be processed in the vector mode. But even circuit elements that use identical models operate in different regions at different points in time, and the different regions are described by different sets of equations, thus hindering vectorization. As a result of these problems, significant overhead penalties are incurred in gathering data into vectors which can utilize the fast vector processing capabilities, and in scattering the results of vector computations back to their ultimate destinations [Cal79, Cal80, Vla82, Ham83, May83, Yam85].

Parallel processors of the MIMD (multiple instruction, multiple data) type consist of separate processors which can perform independent operations on independent sets of operands [Fly72]. The processors can share data either through some type of data communication mechanism or through shared memory. Compared to a vector processor, this type of architecture offers a more flexible environment for exploiting parallelism, because the concurrent operations need not be identical and they need not be performed on operands which are organized as elements of a vector. The use of parallel processors for direct method circuit simulation remains an active area of research. Good performance is easy to obtain in evaluating the models of circuit elements, but effective parallelization of the solution of the sparse unstructured matrix equations is more difficult, due to the large number of fine-grained data dependencies which occur in Gaussian elimination and LU factorization [Win80, Bis86, Cox87, Jac87, Nak87, Sad87, Sma87a].

Relaxation methods for circuit simulation [New84] involve partitioning the circuit into subcircuits which are solved independently, while treating the voltages external to each subcircuit as if they were independent voltage sources. The values of these voltage sources are updated with the solutions of neighboring subcircuits on each iteration of the iterative solution process. These algorithms have natural parallel implementations because of the inherent partitioning of the circuit. Relaxation methods, when carried to convergence, produce precise voltage waveforms of the same quality as standard direct method algorithms. Different forms of relaxation algorithms have been implemented on parallel processors [Gal88], including nonlinear algebraic equation relaxation techniques [Deu84, Web87], waveform relaxation [Whi85b, Uno85, Mat86, Dum87, Sma87b, Sma88b], and waveform-Newton [Sal87b].

Addressed in this thesis is the question of how much of a speed improvement in circuit simulation run time can be obtained by exploiting the natural parallelism of waveform relaxation [Lel82, Whi86, Rue87, Hsi85] on parallel processors. Several different parallel waveform relaxation algorithms are described in Chapter 2, based on the Gauss-Seidel and Gauss-Jacobi (or Jacobi) relaxation methods [Ort70], using window level parallelism and time point pipelining [Whi85b]. The Gauss-Jacobi method in combination with time point pipelining is shown to produce an algorithm with a comparatively high degree of parallelism, which is effective when the number of processors is large compared to the size of the circuit [Sma88b].

The parallel performances of the Gauss-Seidel and Gauss-Jacobi methods are addressed in Chapter 3 from a theoretical viewpoint. A theorem is presented which shows that parallel Gauss-Jacobi is asymptotically faster than parallel Gauss-Seidel when the number of processors is sufficiently large, under certain conditions which apply to the solution of linear equations arising in the simulation of MOS circuits. A formula is also derived for the average ratio of parallelism of the two methods, based on the nonzero structure of the matrix.

In Chapter 4, speedup estimates are computed for a set of 5 benchmark circuits, for the competing parallel waveform relaxation algorithms. The speedups indicate how much faster the algorithms run on multiple processors compared to a single processor. Two categories of speedup estimates are considered. Presimulation estimates are based on simplifying assumptions that allow the estimates to be computed prior to performing a simulation of the circuit. These estimates provide first-order insights into the sources of parallelism, and the factors which inhibit parallelism in real circuit examples. The presimulation estimates also provide a basis for selecting one of the algorithms prior to simulating a particular circuit on a given number of processors. Post-simulation estimates are more accurate than presimulation estimates because they utilize detailed information obtained in the simulation of the circuit on a uniprocessor. Post-simulation estimates are used to generate accurate projections of the potential performance of the algorithms when the number of processors is increased beyond that which is currently available. Post-simulation speedup estimates excluding multiprocessing overhead are compared in subsequent chapters with actual multiprocessor performance results, to determine the extent to which the overhead factors impact the performance. Speedup estimates excluding overhead have been used previously in the study of parallel iterated timing analysis [Deu84].

The parallel waveform relaxation algorithms have been implemented in two programs which run on an Alliant FX/8 multiprocessor, using up to 8 processors. These programs are described in Chapters 5 and 6. Measured performance results are given for the parallel implementations, and these results are compared with the speedup estimates of Chapter 4.

Latency in the computations of waveform relaxation can be exploited to reduce the number of computations that must be performed. Reducing the number of computations reduces the run time on a uniprocessor, and may or may not reduce the run time on a multiprocessor, depending on the data dependencies between the nonlatent computations and on the overhead involved in detecting latency. The impact of latency exploitation on parallel waveform relaxation is addressed in Chapter 7. The primary form of latency that is considered is iteration latency, which has also been called partial waveform convergence in previous work [Whi86]. The impacts of latency on the parallel implementations and on parallel performance are discussed, and performance results are presented. Finally, conclusions are presented in Chapter 8.

CHAPTER 2

PARALLEL WAVEFORM RELAXATION ALGORITHMS

Waveform relaxation is a class of iterative algorithms for solving ordinary differential equations. When applied to the differential equations that describe the voltages of a circuit as a function of time, waveform relaxation is guaranteed to converge, provided that the circuit equations satisfy certain conditions. These conditions are readily satisfied by MOS integrated circuits which are represented by node equations, provided that the inevitable capacitance from each node to ground is included in the equations [New84]. As in other relaxation based algorithms, the equations are partitioned into subsystems which are solved independently during the iterative process. Due to the inherent partitioning of the equations, waveform relaxation exhibits natural parallelism which can be utilized on parallel processors to reduce the overall computation time.

In this chapter, the basic waveform relaxation algorithm is summarized. The form of waveform relaxation implemented in the RELAX2.3 program [Whi85a, Whi86] is used as a model of the basic algorithm on a uniprocessor. Different approaches for exploiting parallelism are then identified. Both the Gauss-Jacobi and Gauss-Seidel relaxation methods are considered, and the techniques of exploiting parallelism at the window level or at the time point level are compared. Task graphs representing the computations and their interdependencies are introduced. A first-order analysis of the parallel algorithms is presented based on the task graphs and some simplifying assumptions. The combination of the Gauss-Jacobi relaxation method and time point pipelining is identified as the algorithm with the greatest potential parallelism of the algorithms considered. The performance of the parallel algorithms is investigated in greater detail in subsequent chapters.

2.1 Waveform Relaxation

Consider a circuit with N nodes, which satisfies a system of differential node equations of the form

$$\dot{q}(v(t), u(t)) = f(v(t), u(t)), \quad t \in [0, t_f] \tag{2.1}$$

with initial conditions

$$v(0) = V, \tag{2.2}$$

where $t \in \mathbb{R}$ is time, $v : \mathbb{R} \to \mathbb{R}^N$ is the vector of node voltages, $V \in \mathbb{R}^N$ is the vector of initial node voltages, $u : \mathbb{R} \to \mathbb{R}^r$ is the vector of known source values, $f : \mathbb{R}^N \times \mathbb{R}^r \to \mathbb{R}^N$ is the vector of currents flowing into charge storage elements at each node, $q : \mathbb{R}^N \times \mathbb{R}^r \to \mathbb{R}^N$ is the vector of charges stored at the nodes, and $\dot{q}$ is the time derivative of q. Prior to solving the equations by waveform relaxation, the system of equations must be partitioned into subsystems. Since (2.1) is written in terms of node equations, the equation partitioning problem is equivalent to partitioning the set of circuit nodes into subsets that are mutually exclusive and exhaustive. Each subset of nodes, together with the circuit elements connected to the nodes, represents a subcircuit. To achieve fast convergence speed during the relaxation iterations, it is important to keep nodes that are tightly coupled to each other in the same subcircuit.

For the Gauss-Seidel relaxation method, an ordering of the subcircuits must be defined. For fast convergence, this ordering should reflect the predominant direction of signal flow in the circuit. If the circuit contains n subcircuits, then the subcircuits are assigned numbers from 1 to n such that, in as many cases as possible, strong signals flow from lower numbered subcircuits to higher numbered subcircuits. When feedback paths are present, some signals will flow from higher numbered subcircuits to lower numbered subcircuits, and these signal paths can lead to slow convergence.

To counteract the negative impact of global and local feedback on convergence speed, the time interval $[0, t_f]$ is partitioned into subintervals called *windows*. Convergence occurs more rapidly at the beginning than at the end of a window. Consequently, if the size of a window is reduced, in order to include only the first part of the original window, then fewer iterations are required for convergence. However, if the window sizes are made too small, then excessive time points will be introduced at the window boundaries for those subcircuits that have constant or slowly changing signals. The window boundaries are modified dynamically during the solution process based on the observed convergence speed and signal activity.

During the actual equation solution phase of a waveform relaxation program, the windows are processed sequentially, with the final solution point of one window serving as the initial condition of the next window. Within each window, the subcircuits are solved independently on each iteration, using previously computed or guessed values for the voltages which are external to the subcircuit being solved. The differential equation of subcircuit i in window $[t_a, t_b]$ on iteration k is given by

$$\dot{q}_i(\hat{v}_{1,i}^{(k)}, \ldots, \hat{v}_{i-1,i}^{(k)}, v_i^{(k)}, \hat{v}_{i+1,i}^{(k)}, \ldots, \hat{v}_{n,i}^{(k)}, u) =$$
$$f_i(\hat{v}_{1,i}^{(k)}, \ldots, \hat{v}_{i-1,i}^{(k)}, v_i^{(k)}, \hat{v}_{i+1,i}^{(k)}, \ldots, \hat{v}_{n,i}^{(k)}, u),\ t \in [t_a, t_b], \tag{2.3}$$

where v_i is the vector of node voltages for those nodes that are in subcircuit i, and q_i and f_i are vector functions containing the components of q and f that correspond to the nodes of subcircuit i. The vector $\hat{v}_{j,i}^{(k)}$, for any $j \neq i$, is a vector of voltage waveforms for the nodes belonging to subcircuit j, which are treated as known source waveforms in the solution of subcircuit i. The iteration from which these waveforms are obtained depends on which relaxation method is used. When Gauss-Seidel relaxation is used,

$$\hat{v}_{j,i}^{(k)} = \begin{cases} v_j^{(k)}, & \text{if } j < i \\ v_j^{(k-1)}, & \text{if } j > i \end{cases} , \tag{2.4}$$

and when Gauss-Jacobi relaxation is used,

$$\hat{v}_{j,i}^{(k)} = v_j^{(k-1)}, \text{ for all } j \neq i. \tag{2.5}$$

In all cases $v^{(0)}$ is a vector of initial guess waveforms.

The basic waveform relaxation algorithm with windowing, using either Gauss-Seidel or Gauss-Jacobi relaxation can be summarized as follows:

Algorithm 2.1. Windowed Waveform Relaxation

```
partition into subcircuits
order subcircuits
$t_a \leftarrow 0$
while ($t_a < t_f$) {                         /* window loop */
        choose $t_b$
        $k \leftarrow 1$
        repeat {                              /* relaxation iteration loop */
                if (any subcircuit used too many time points) decrease $t_b$
                if ($k \in K$) {
                        decrease $t_b$
                        decrease integration error tolerance
                }
                for ($i = 1, 2, ..., n$) {    /* subcircuit loop */
                        solve (2.3) for $v_i^{(k)}$    /* subcircuit evaluation task */
                }
                $k \leftarrow k + 1$
        } until (convergence obtained)
        $t_a \leftarrow t_b$
        reinitialize integration error tolerance
}
```

The endpoint of the current window, t_b, is initially determined based on the number of points and the number of iterations of the previous window. Target values of approximately 60 time points and 5 iterations are used in RELAX2.3 to control the window sizes. If these targets are exceeded, the window size is decreased. If the number of time points and iterations in a window are significantly below the target values, then the size of the following window is increased. Most windows converge without triggering the window reduction mechanism.

The set K contains the iteration numbers on which the window size is to be reduced to encourage faster convergence. RELAX2.3 uses $K = \{6, 12, 18, \cdots\}$. The reduction of the integration error tolerance which accompanies a window reduction causes smaller time steps to be used in the numerical integration, since the waveform relaxation convergence theorem assures convergence only when the step size is sufficiently small [Whi86]. Convergence is detected on iteration k if $v^{(k)}$ matches $v^{(k-1)}$, within a specified convergence tolerance, at each point in the window $[t_a, t_b]$.

Algorithm 2.1 does not include the partial waveform convergence feature in which subcircuit evaluations are bypassed if the input waveforms of the subcircuit match the previous iteration. The use of partial waveform convergence in parallel waveform relaxation is addressed in Chapter 7.

Nearly all of the computational effort is concentrated in solving (2.3) in the innermost loop of the algorithm. This is a problem of exactly the same form as the original problem represented by (2.1); however, the size of the problem is smaller both in terms of the number of unknowns and the length of the time interval. Conventional circuit simulation techniques [Chu75, Nag75] are used to solve the subcircuit equations: the time scale is discretized by an implicit, stiffly stable, variable step size numerical integration algorithm [Gea71]; the nonlinear algebraic equations which result at each time point are solved iteratively using Newton's method; and the linear equations arising on each Newton iteration are solved using Gaussian elimination.

2.2 Parallel Algorithms

The objective of this section is to identify the parallelism in Algorithm 2.1, which can be exploited on parallel processors. Attention is restricted to the portion of the algorithm that actually solves the differential equations, excluding the partitioning and ordering steps. Although parallelism can also be exploited in partitioning and ordering, these tasks account for a small fraction of the overall computation time, and therefore, the parallelization of these tasks offers only limited opportunity for speeding up the overall run time.

At the highest level of Algorithm 2.1 is the window loop. In each window, the initial conditions are obtained from the final voltage values of the previous window. These values are known only when the previous window converges. Consequently, the windows must be processed serially. Within a window, the main computational tasks are *subcircuit evaluation tasks*. Each of these tasks consists of the computations required to solve a subcircuit over an entire time window on one relaxation iteration. Some of the subcircuit evaluation tasks can be performed concurrently, but restrictions are imposed on the sequence of task executions due to the propagation of waveforms from one task to another.

A conservative method of managing the restrictions on parallel task executions is the *full window technique*. In this scheme, a subcircuit evaluation task is allowed to begin executing only after all of its input waveforms from other tasks are available over the entire current window. For example, if subcircuit 7 on iteration 3 requires input waveforms from subcircuit 1 on iteration 3 and subcircuit 8 on iteration 2, then the evaluation of subcircuit 7 would not begin on iteration 3 until after the solutions of subcircuits 1 and 8 were completed over the entire window for iterations 3 and 2, respectively. Scheduling of subcircuit evaluation tasks in the full window technique is relatively inexpensive. When a task finishes, it can check to see if any of the tasks to which it supplies waveforms are ready to start executing. These checks need only be performed after a task computes the last time point of a window.

The full window technique utilizes a course grain of parallelism in that it treats a subcircuit evaluation task as an indivisible schedulable entity, and subcircuit evaluation tasks normally involve a large number of computations. The advantage of this approach is that the scheduling overhead is small compared to the amount of computations performed in the subcircuit evaluations. The disadvantage is that the full degree of available parallelism is not exploited. The *time point pipelining* parallelization strategy exposes greater parallelism, at the expense of increased overhead, by using a finer granularity.

In time point pipelining, the subcircuit evaluation tasks are broken down into subtasks, each consisting of the evaluation of a subcircuit on a single iteration at a single time point. Each of these subtasks is allowed to begin executing as soon as all of its input data are available. In the example cited above, subcircuit 7 may compute a time point at time t after the evaluations of subcircuits 1 and 8 have progressed through time t on iterations 2 and 3, respectively. Since computations can begin sooner in time point pipelining than in the full window technique, the overall completion time should be smaller for time point pipelining, and the degree of parallelism should be larger.

The scheduling of computations in time point pipelining is more expensive than in the full window technique, because checks must be performed after the computation of each time point to determine if any other subtask is eligible to execute as a result of the availability of the newly computed time point data. Consequently, the full window technique and time point pipelining offer a tradeoff between lower overhead and greater parallelism.

Further parallelism can be exploited within each individual time point subtask, with an accompanying increase in overhead. Since standard direct method circuit simulation techniques are used at the subcircuit level, the problem of parallelizing computations within a *single* subcircuit evaluation task is equivalent to the problem of parallelizing the standard direct methods. The evaluation of the model equations for each circuit element can be performed in parallel on each Newton iteration, and the loading of the Jacobian matrix is readily parallelized. The parallelization of the solution of the linear equations is hampered by the high degree of sparsity in the matrix, the irregular pattern of nonzero matrix entries, and the large number of data dependencies in Gaussian elimination. In the context of waveform relaxation, the amount of parallelism available within a single subcircuit evaluation task will be small compared to the parallelism of direct methods applied to the entire circuit, because the subcircuit sizes are typically small. For those circuits which cannot be successfully partitioned into subcircuits of uniformly small size, the use of parallelism within the subcircuit evaluation tasks offers a potential for accelerating the solution of the larger subcircuits, which tend to create bottlenecks in the full window technique and time point pipelining. The use of parallelism within individual subcircuit evaluation tasks is not considered further here. Instead, attention is focused on the natural parallelism between different subcircuits that arises directly from the use of waveform relaxation.

The full window technique and time point pipelining are two different methods of orchestrating the parallel execution of a fixed set of computations. Contrastingly, the Gauss-Seidel and Gauss-Jacobi relaxation methods result in different computations being performed to reach approximate solutions that match the exact solution within some acceptable tolerance. The Gauss-Seidel method generates a set of computations which generally converges to the solution in fewer iterations than Gauss-Jacobi. However, the Gauss-Jacobi method generates a set of computations which has a higher degree of parallelism, because all subcircuit evaluation tasks of any given iteration can be executed concurrently without any waveform communications between the tasks. This raises the question of whether the extra parallelism of Gauss-Jacobi is sufficient to overcome the penalty of requiring a larger number of iterations.

By combining either the full window technique or time point pipelining with either the Gauss-Seidel or Gauss-Jacobi relaxation method, one of four different parallel waveform relaxation algorithms is obtained. The four algorithms and their principal tradeoffs are summarized in Fig. 2.1.

2.3 Task Graphs

Task graphs are useful tools in studying and implementing parallel algorithms. A task graph is a directed graph in which the vertices represent tasks and the arcs represent precedence constraints [Hwa84]. An arc from vertex i to vertex j indicates that task i must finish before task j is allowed to begin executing; i is said to be a predecessor of j and j is a successor of i.

Waveform relaxation task graphs are closely related to the subcircuit interconnection structure, which is conveniently represented by a subcircuit graph. For a given circuit partitioned into n subcircuits, in which the subcircuits are numbered from 1 to n

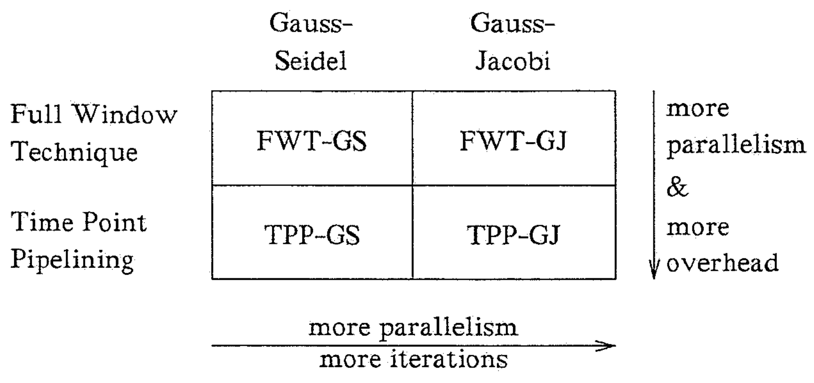


Figure 2.1. Parallel waveform relaxation algorithms.

reflecting a given ordering of the subcircuits, the subcircuit graph is defined as follows:

Definition 2.1. *The subcircuit graph G contains one vertex for each subcircuit, labeled with the subcircuit number. An arc exists from vertex i to vertex j if and only if a node equation in subcircuit j depends on a node voltage which belongs to subcircuit i.*

For the Gauss-Seidel method, waveforms are propagated differently depending on whether the waveform is computed in a subcircuit that has a higher or lower number than the destination subcircuit, as indicated in (2.4). This distinction motivates the next definition.

Definition 2.2. *An arc in G from i to j is called a feedforward arc if $i < j$, and is called a feedback arc if $i > j$.*

This terminology agrees with the standard circuit concept of feedback provided the ordering of subcircuits conforms to the predominant direction of signal flow in the circuit.

2.3.1 Full window technique

A waveform relaxation task graph for the full window technique depends on which relaxation method is employed and on the number of iterations to be performed, as reflected in the notation introduced in the next definition.

Definition 2.3. *For a given G, the task graphs for m iterations of the Gauss-Jacobi and Gauss-Seidel methods using the full window technique are denoted as $T_{GJ,m}$ and $T_{GS,m}$, respectively. Each vertex represents a subcircuit evaluation task and is labeled with an ordered pair (k, i), where k is the iteration number and i is the subcircuit number. An arc exists from (k_1, i_1) to (k_2, i_2) if and only if (k_2, i_2) requires an input waveform from (k_1, i_1).*

The task graphs can be constructed from G, based on the waveform communication rules for the applicable relaxation method. Each vertex i of G maps to m vertices in the task graph, labeled (k,i), for $k \in \{1,2, \cdots ,m\}$. Each of these task graph vertices is said to be an *instance* of vertex i in G. For the Gauss-Jacobi task graph, each arc in G from a vertex i to a vertex j maps to $m-1$ arcs in $T_{GJ,m}$, from (k,i) to $(k+1, j)$, for $k \in \{1,2, \cdots ,m-1\}$. These arcs are said to be instances of the arc in G from i to j. For the Gauss-Seidel task graph, each feedback arc in G from i to j maps to $m-1$ instances in $T_{GS,m}$, from (k,i) to $(k+1, j)$, for $k \in \{1,2, \cdots ,m-1\}$; and each feedforward arc from i to j maps to m instances in $T_{GS,m}$, from (k,i) to (k, j), for $k \in \{1,2, \cdots ,m\}$. Figure 2.2 shows a sample subcircuit graph and corresponding Gauss-Seidel and Gauss-Jacobi task graphs for the case $m=2$.

The instance relationships introduced in the construction of the task graphs define mappings between elements of the subcircuit graph and the task graphs. These relationships are useful in proving theorems which relate properties of parallel relaxation methods to properties of G. If T is one of the task graphs, then observe that any directed path in T maps to a directed walk in G, such that the j^{th} vertex (or arc) of the path in T is an instance of the j^{th} vertex (or arc) of the walk in G. An arc in T is referred to as a feedback or feedforward arc, based on whether it is an instance of a feedback or feedforward arc, respectively. To simplify subsequent discussions of the subcircuit and task graphs, the terms *path*, *walk*, and *cycle* will be used to refer to *directed* paths, walks, and cycles.

2.3.2 Time point pipelining

Task graphs for the time point pipelining method can be constructed in which each vertex represents the computation of a single time point in a single subcircuit on a single iteration.

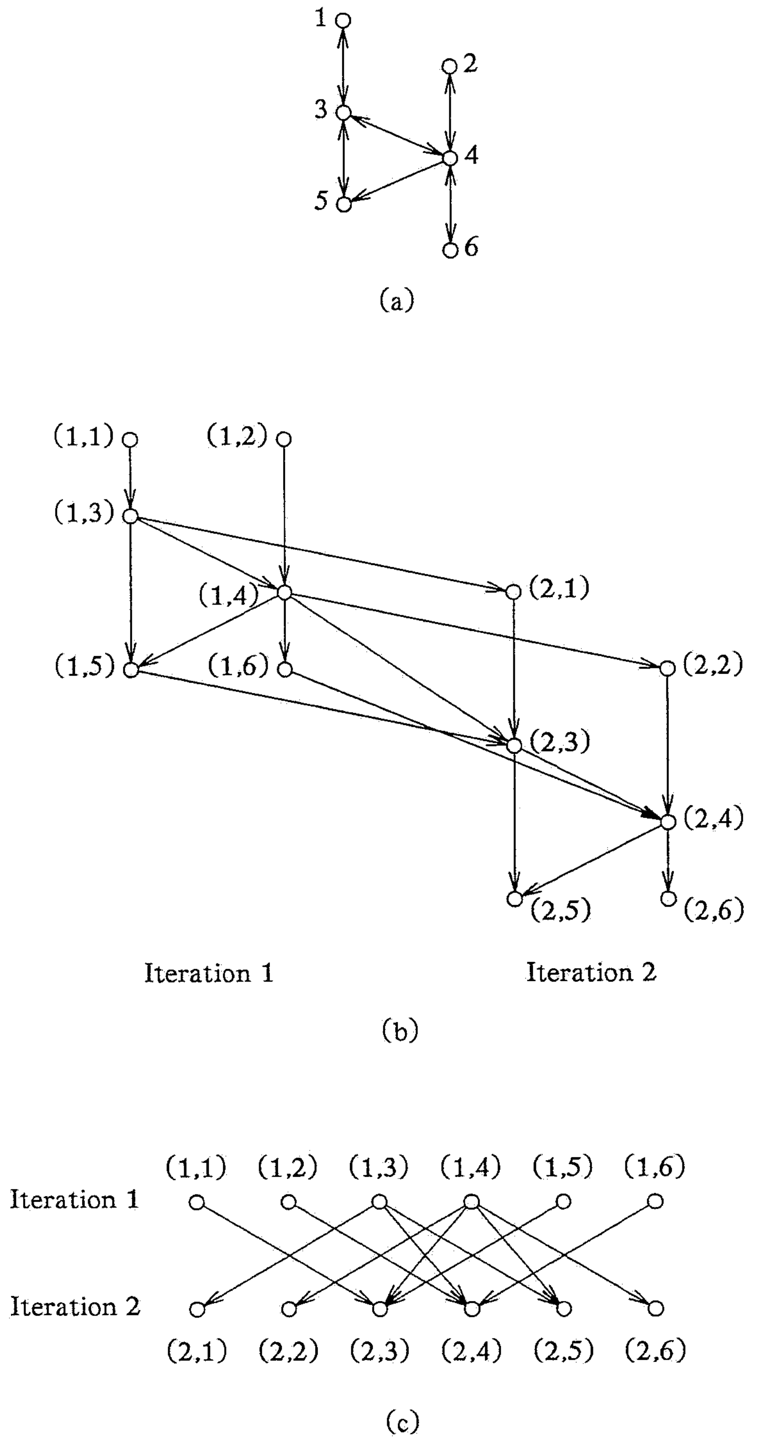


Figure 2.2. Subcircuit graph and related task graphs: (a) G ; (b) $T_{GS,2}$; (c) $T_{GJ,2}$.

Definition 2.4. *For a given circuit with a given G, the unaugmented task graphs for m iterations of the Gauss-Jacobi and Gauss-Seidel methods using time point pipelining are denoted as $T_{TPP\text{-}GJ,m}$ and $T_{TPP\text{-}GS,m}$, respectively. Each vertex represents a time point subtask which consists of the evaluation of a time point in a subcircuit on a relaxation iteration, and is labeled (k,i,t), where k is the iteration number, i is the subcircuit number, and t is the value of the time variable.*

A time point pipelining task graph T_{TPP} can be constructed from the corresponding full window technique task graph T and from knowledge of the number and locations of the time points. When a time point at some time t is computed in subcircuit evaluation task (k,i), its time value is initially determined based on the previous time step and the estimated local truncation error at that step. Let $t_{init}(k,i,t)$ be this initial choice of the time value. If the number of Newton iterations is excessive or if the local truncation error is excessive for the new time point at time $t_{init}(k,i,t)$, then the time step is reduced repeatedly until the Newton iteration count and the truncation error are acceptable. The final value of the time variable t for the time point is less than or equal to $t_{init}(k,i,t)$. Consequently, when computing the time point at time t, it is necessary for the input waveforms to be available through time $t_{init}(k,i,t)$.

The construction of T_{TPP} begins by defining vertices for all the time point subtasks and labeling them as specified in Definition 2.4. Consider any arc in T from a task (k_1,i_1) to (k_2,i_2). For each time point t_2 in (k_2,i_2), let t_1 be the smallest time point in (k_1,i_1) such that $t_1 \geqslant t_{init}(k_2,i_2,t_2)$. Then there is an arc in T_{TPP} from (k_1,i_1,t_1) to (k_2,i_2,t_2).

Unlike the task graphs for the full window technique, the time point pipelining task graphs cannot be constructed solely on the basis of G and m. The number of time points to be computed and their locations on the time axis depend on the signal activity in the subcircuits, and this information is not known prior to simulating the circuit. The time point pipelining task graphs may be constructed after the simulation is completed, or they can be constructed dynamically during the simulation. For this reason, the time point pipelining task graphs play a less important role than the full window technique task graphs in presimulation studies of parallelism. Due to the relationship between the time point pipelining and full window technique task graphs, the simpler full window technique task graphs can be used in the study and implementation of time point pipelining by recognizing implicitly that each task consists of a sequence of time point subtasks with suitable precedence constraints.

2.4 First Order Analysis of Parallel Algorithms

Based on the task graphs and some simplifying assumptions, first-order measures of the parallelism of different relaxation methods and different parallelization strategies can be derived. The first-order estimates are not accurate measures of the parallel processing speedups that can be obtained in practice. Nonetheless, these estimates provide basic insights into the different parallel algorithms and the fundamental tradeoffs involved. More accurate models of parallelism are developed in Chapter 4.

2.4.1 Full window technique

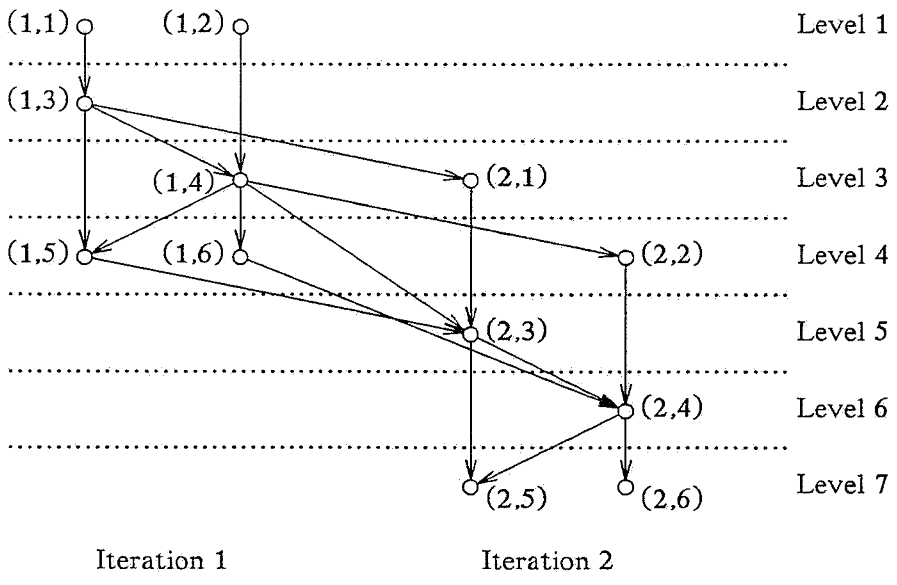
The following simplifying assumptions apply to the first-order analysis of the full window technique:

(a) Each subcircuit evaluation task requires exactly one unit of computation time.

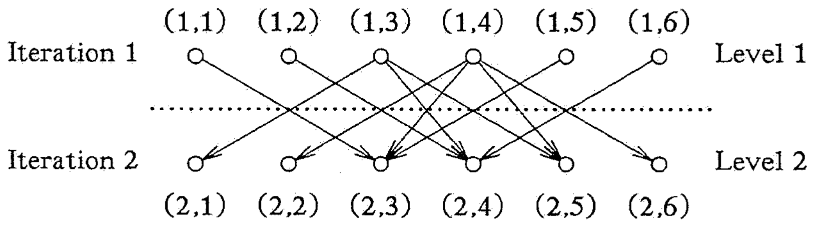
(b) There is a sufficient number of processors such that each task can begin executing immediately after all of its predecessors in the task graph have finished executing.

(c) There is no parallel processing overhead due to task scheduling, data communications, or contention for shared resources.

Since each task is assumed to require one unit of time, and since each task is assumed to begin immediately after its predecessors terminate, the task graph can be partitioned into levels such that the tasks in level l are those which are active during processor time interval $(l-1, l]$. Figure 2.3 shows the levelizations of the task graphs of Fig 2.2.



(a)



(b)

Figure 2.3. Levelized task graphs: (a) $T_{GS,2}$; (b) $T_{GJ,2}$.

For a task graph T, the time required to execute all the tasks on a multiprocessor with a sufficient number of processors is equal to the depth of the graph, $D(T)$, which is defined as the number of vertices in a longest directed path, or equivalently the number of levels in the levelized graph. The average number of tasks executed at each step is the average width of the graph, $W(T)=|V(T)|/D(T)$, where $|V(T)|$ is the total number of vertices in the graph. For the example in Fig. 2.3, $D(T_{GS,2})=7$, $D(T_{GJ,2})=2$, $W(T_{GS,2})\approx 1.7$, and $W(T_{GJ,2})=6$.

The superior parallelism of Gauss-Jacobi over Gauss-Seidel is apparent from the task graphs. In general, m iterations of Gauss-Jacobi require only m steps, since within each iteration all the tasks can be executed concurrently. Except for some degenerate cases mentioned in Chapter 3, $D(T_{GJ,m})=m$ and $W(T_{GJ,m})=n$. In contrast, the tasks of any single iteration of Gauss-Seidel generally have data interdependencies such that they cannot all be executed concurrently. Normally, $D(T_{GS,m})>>D(T_{GJ,m})$ and $W(T_{GS,m})<<W(T_{GJ,m})$.

The superior parallelism of Gauss-Jacobi is achieved at the expense of an increase in the number of iterations required to converge. In the Gauss-Seidel case a signal can propagate through an entire forward signal path in a single iteration. In the Gauss-Jacobi case, however, signals propagate at a rate of one subcircuit per iteration. This suggests that circuits with long signal paths will require a large, perhaps prohibitive, number of Gauss-Jacobi iterations. However, in a given time window, long signal paths are frequently broken into smaller subpaths by logic gates, pass transistors, or clocked devices which block the signal flow. The number of Gauss-Jacobi iterations is then determined by the number of subcircuits in each subpath and by feedback between subcircuits. Even if a long path is present, the Gauss-Jacobi iterations may not be excessive if the subpaths are short.

Consider the subcircuit graph and related task graphs in Fig. 2.4. The circuit consists of a single long path with no feedback. The Gauss-Seidel method produces the exact solution in 1 iteration, and the Gauss-Jacobi method requires exactly n iterations. However, both methods require exactly n units of computation time. The Gauss-Jacobi

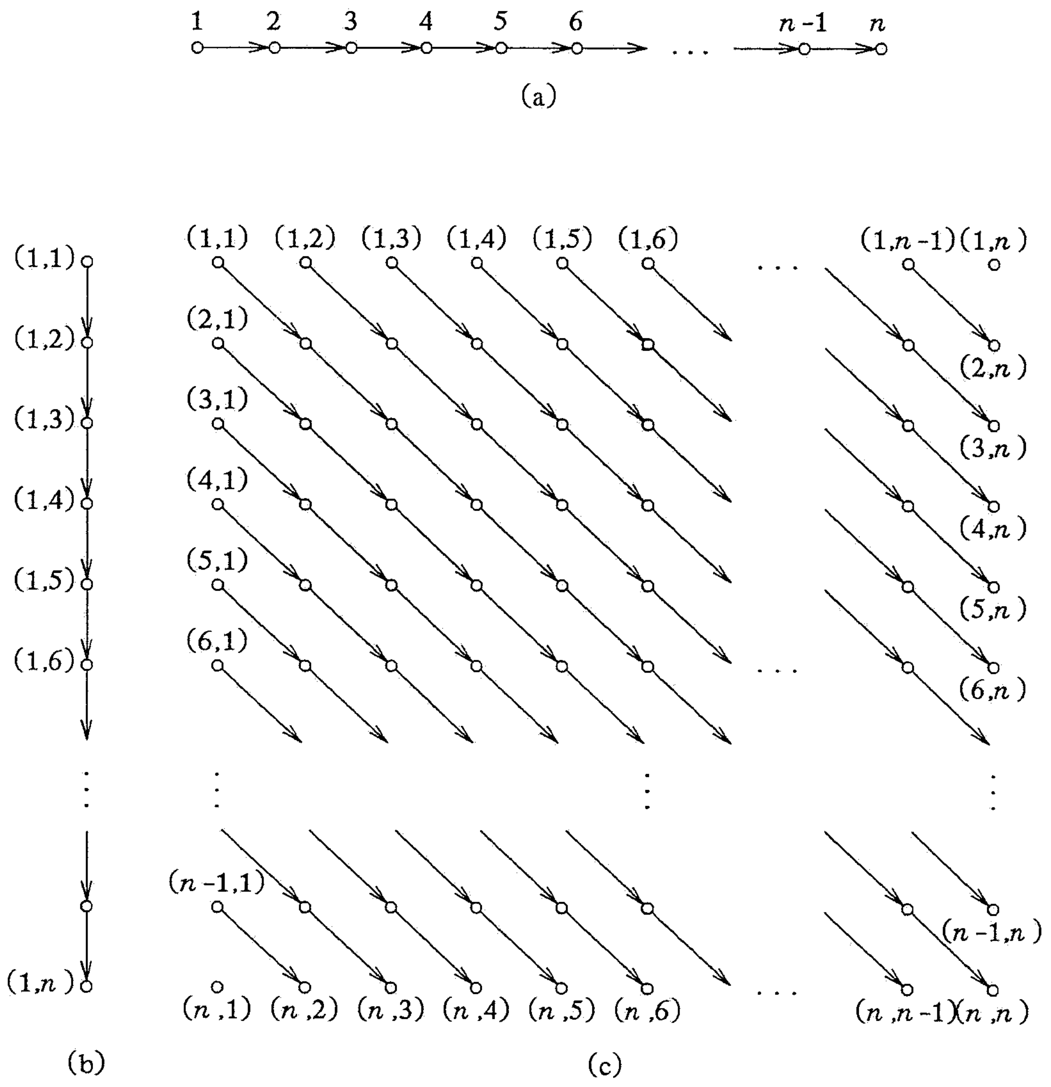


Figure 2.4. Long signal path example. (a) G ; (b) $T_{GS,1}$; (c) $T_{GJ,n}$.

method requires more processors, but it produces the same result in the same amount of time in this example.

Suppose that a logic gate or pass transistor in subcircuit $n/2$ is in a state throughout some particular time window such that signal flow from subcircuit $n/2$ to $n/2+1$ is blocked. Since the subcircuit graph is unchanged, the Gauss-Seidel method will still require 1 iteration and n units of time. The Gauss-Jacobi method will obtain the exact solution in $n/2$ iterations, requiring only $n/2$ units of time, because the longest effective signal path has length $n/2$. Gauss-Jacobi effectively solves the two isolated portions of the circuit concurrently.

In digital circuits, signal paths are frequently blocked by clocked devices during certain intervals of time. The automatic windowing algorithm of RELAX2.3 typically chooses windows which are smaller than one clock period. Therefore, in a typical window, the effective signal path lengths are much shorter than the paths in the subcircuit graph. The Gauss-Jacobi method automatically exploits the parallelism between the essentially disconnected portions of the circuit; the Gauss-Seidel method does not.

In the segmented waveform relaxation method [Dum87], the time axis is partitioned into time segments, whose boundaries are explicitly synchronized with clock transitions. Portions of the circuit which are disconnected by clocked logic elements during a time segment are recognized, and the Gauss-Seidel method is applied separately and concurrently to the disconnected portions for the first iteration of the relaxation. This approach exploits the parallelism between different parts of the circuit while retaining the Gauss-Seidel ordering within the isolated circuit portions. However, the program is required to recognize the clocked elements and derive the resulting effective circuit partitioning. Thus, the method is less general and more complicated than Gauss-Jacobi. In those cases where it is applicable, it may be more effective than

Gauss-Jacobi when the number of processors is limited, because it avoids some of the redundant computations which tend to occur in Gauss-Jacobi on the iterations before a signal reaches a particular subcircuit.

The preceding discussion gives reasons why the Gauss-Jacobi method may be preferable to Gauss-Seidel on parallel processors. Further theoretical and experimental evidence will be presented in subsequent chapters to show that when a sufficient number of processors is available, parallel Gauss-Jacobi is faster than parallel Gauss-Seidel.

In preparation for comparing the full window technique with time point pipelining, the following assumption is added to those previously introduced for the first-order analysis of the full window technique:

(d) The number of tasks in each level of the task graph is constant.

The number of active tasks can then be plotted as a function of time, as in Fig. 2.5(a). Since a different number of iterations is generally required for the Gauss-Jacobi and Gauss-Seidel methods, the symbols m_{GJ} and m_{GS} are used to represent the number of iterations required for each method to converge, where typically $m_{GJ} > m_{GS}$. The curves emphasize the fact that even though Gauss-Jacobi requires more computations as represented by the area under the curve, its greater parallelism can lead to a smaller overall computation time. The number of processors required to achieve the fastest possible computation time for Gauss-Jacobi is greater than that needed for Gauss-Seidel, as indicated by the heights of the curves.

2.4.2 Time point pipelining

The first-order analysis of time point pipelining is based on the full window task graphs. The tasks are implicitly divided into subtasks, each consisting of the evaluation

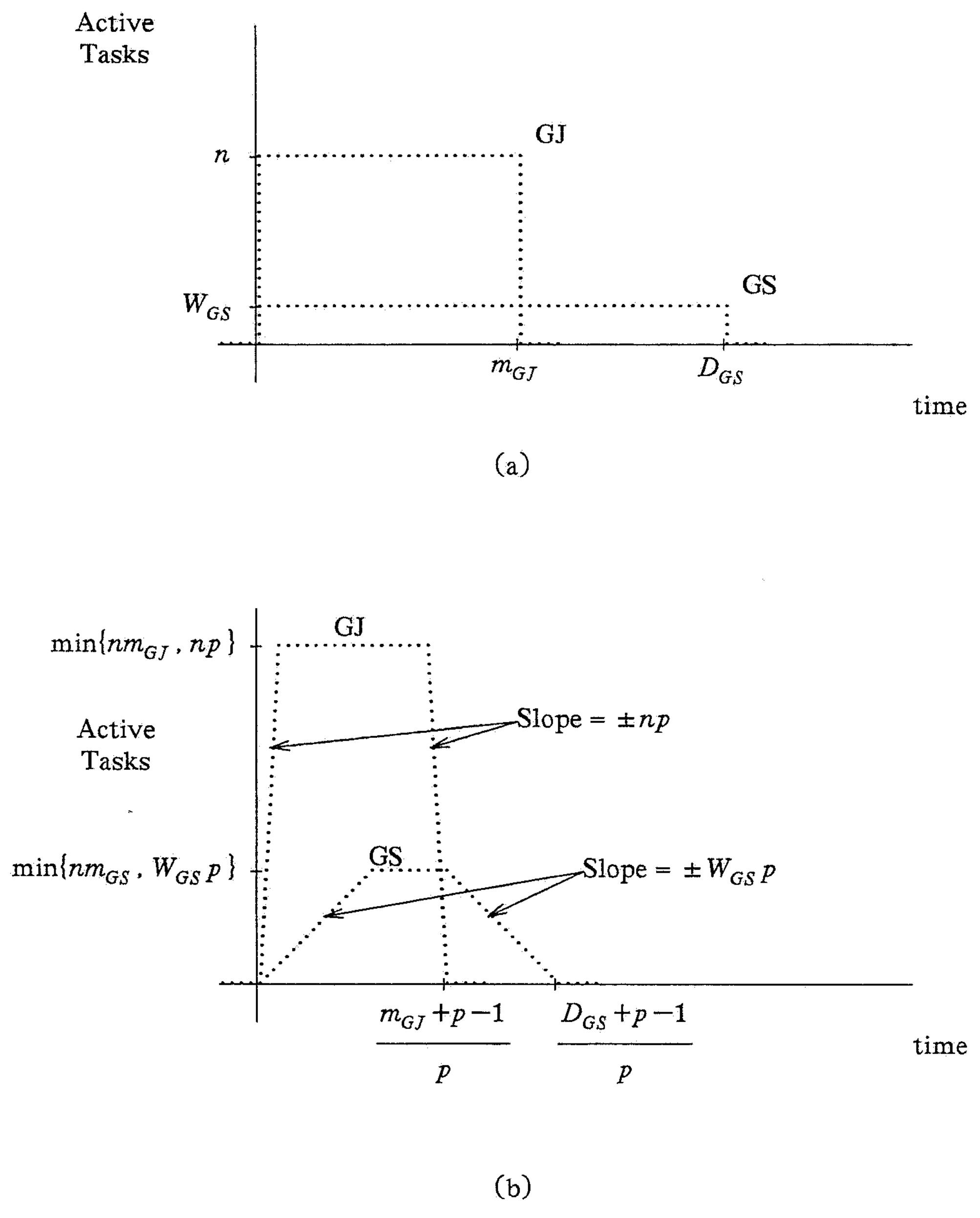


Figure 2.5. Active tasks vs. time for (a) full window technique and (b) time point pipelining, based on simplified parallelism model. $D_{GS} \equiv D(T_{GS,m_{GS}})$ and $W_{GS} \equiv W(T_{GS,m_{GS}})$.

of a subcircuit on an iteration at a time point. Subtasks are represented by ordered triples of the form (k, i, t), such that (k, i) is the subcircuit evaluation task which contains the subtask, and t is the value of the time variable at the time point. The following simplifying assumptions apply to the first-order analysis of time point pipelining, in addition to the assumptions used in the full window case:

(e) Each subcircuit evaluation task contains p time point subtasks.

(f) Each subtask requires the same amount of computation time, $1/p$ time units.

(g) The time point locations on the time axis are identical in each task.

(h) No time step reductions occur due to excessive integration truncation error or excessive Newton iterations.

(i) There is a sufficient number of processors such that each subtask (k, i, t) can begin execution immediately after all predecessors of (k, i) have finished computing the time point at time t, and (k, i) has computed the time point preceding t.

These assumptions are generally not accurate, but are useful to demonstrate the basic properties of the algorithms. Assumptions (e) and (g) do not hold exactly in practice because the time points are chosen independently in each subcircuit based on the signal activity within the subcircuit. Assumption (f) does not hold exactly because subcircuits of different sizes have subtasks requiring different computation times. Generally the assumptions tend to result in overestimates of parallelism.

When time point pipelining was originally introduced it was used in conjunction with the Gauss-Seidel method as a way to increase the parallelism without giving up the fast convergence properties of Gauss-Seidel. However, the technique is equally applicable to any relaxation method that can be represented by a full window technique task graph, including Gauss-Jacobi.

Let T be any full window technique task graph, and consider the operation of the time point pipelining algorithm applied to T as the parallel computation time l progresses, using the stated simplifying assumptions. Let $c(l)$ be the number of active tasks in processor time interval $((l-1)/p, l/p]$. Initially, each task in level 1 of T can compute its first time point, and therefore $c(1)=W(T)$, assuming the task graph has constant width. Once these time point computations are completed, the second time points of the level 1 tasks can be computed concurrently with the first time points of the level 2 tasks, resulting in $c(2)=2W(T)$. The parallelism continues to increase at a rate of $W(T)$ until either all tasks in the task graph are active simultaneously, or until the last time point is computed in the level 1 tasks. Consequently, the peak parallelism is

$$\max_{l}\{c(l)\} = \min\{nm, W(T)p\}. \tag{2.6}$$

After the last time point is computed in the level 1 tasks and after the first time point is computed in the tasks of the final level, the parallelism decays at a rate of $W(T)$, because at each step another level completes its last time point and no new levels are started. The final completion time is given by $[D(T)+p-1]/p$, because after $D(T)$ steps the bottom level tasks complete their first time points, and after $p-1$ more steps they complete their last time points, where each step requires $1/p$ time units. Combining these results, a formula for $c(l)$ is obtained for all integers l:

$$c(l) = \begin{cases} lW & , 0<l \leqslant \min\{D, p\} \\ \min\{nm, pW\} & , \min\{D, p\}<l \leqslant \max\{D, p\} \\ (D+p-l)W & , \max\{D, p\}<l \leqslant (D+p-1) \\ 0 & , \text{otherwise} \end{cases} , \tag{2.7}$$

where D and W are the depth and width of the full window technique task graph.

Generic plots of the number of active tasks as a function of parallel processor time, based on (2.7), are shown in Fig. 2.5(b) (p. 25) for Gauss-Jacobi and Gauss-Seidel. The substitution $W(T_{GJ,m})=n$ has been used in the figure to emphasize the fact that typically $W(T_{GJ,m_{GJ}})=n >> W(T_{GS,m_{GS}})$. Note that the parallelism at each point using time point pipelining is at least as large as the full window parallelism using the same relaxation method. Also note that both the peak parallelism and the rate of growth of parallelism are greater for Gauss-Jacobi time point pipelining than for Gauss-Seidel time point pipelining. Finally, the number of processors required to use all the available parallelism of Gauss-Jacobi time point pipelining can be quite large for a large circuit if large windows are used.

2.5 Augmented Task Graphs

The task graphs $T_{GJ,m}$ and $T_{GS,m}$ account for the evaluation of subcircuits and the data communications between subcircuit evaluation tasks. Although these are the most important tasks and data dependencies, they are not the only ones. Computations must be performed to determine when the iterations converge. The convergence checking operations can either be packaged inside the subcircuit evaluation tasks, or they can be treated as separate tasks. In either case, the task graph must be augmented by adding arcs or both vertices and arcs, to account for the additional data dependencies and, in the latter case, the additional tasks.

2.5.1 Separate convergence checking tasks

For each subcircuit evaluation task (k,i) in window $[t_a,t_b]$, a convergence checking task $(k,i)_{cc}$ can be defined, which executes the following algorithm:

Algorithm 2.2. Convergence Checking Task: $(k,i)_{cc}$

```
    if ($v_i^{(k)}(t)$ matches $v_i^{(k-1)}(t)$, for all $t \in [t_a,t_b]$, within tolerance) {
        $unconv_k \leftarrow unconv_k - 1$
        if ($unconv_k$ =0) signal global convergence
}
```

The counter $unconv_k$ represents the number of unconverged subcircuits in iteration k, and is initialized to n. The decrementing and testing of the counter must be performed in a critical section to insure the integrity of the count values since different convergence checking tasks may execute concurrently. When global convergence is signaled, then any remaining incomplete tasks in the task graph are superfluous since they are for iterations greater than the iteration in which convergence was obtained.

The addition of the convergence checking tasks and the accompanying data dependencies are reflected in the augmented task graphs which are defined as follows:

Definition 2.5. *For any task graph $T_{GJ,m}$ or $T_{GS,m}$, the corresponding augmented task graph $\hat{T}_{GJ,m}$ or $\hat{T}_{GS,m}$ is constructed by adding a task $(k,i)_{cc}$ for each task (k,i), and adding arcs from (k,i) to $(k,i)_{cc}$ for all $k \in \{1,...,m\}$, $i \in \{1,...,n\}$, and adding arcs from $(k-1,i)$ to $(k,i)_{cc}$ for all $k \in \{2,...,m\}$, $i \in \{1,...,n\}$.*

An important feature of these augmented task graphs is that there are no new arcs terminating at subcircuit evaluation tasks. Consequently, the parallel completion time of the subcircuit evaluation tasks is not affected by the additions to the task graph. Convergence checking tasks can be executed concurrently with subcircuit evaluation tasks, except for the convergence checking task corresponding to the last subcircuit evaluated in the converging iteration. Therefore, the parallel completion time for the augmented graph is greater than the original graph by the time of one convergence checking task, which is much smaller than a subcircuit evaluation task. For this reason, the use of the unaugmented task graphs is justified in the study of parallel waveform relaxation, even though these graphs do not explicitly account for convergence checking.

2.5.2 Convergence checking in subcircuit evaluation tasks

An alternative approach to handling convergence checking is to treat the convergence checking task $(k,i)_{cc}$ as part of the subcircuit evaluation task (k,i). This results in a simpler implementation with lower overhead for task scheduling. However, extra arcs must be added to the task graph to assure that the waveforms are available from $(k-1,i)$ as needed by the convergence checker inside (k,i). A different type of augmented task graph results from this treatment of convergence checking, as given in the following definition.

Definition 2.6. *For any task graph $T_{GJ,m}$ or $T_{GS,m}$, the corresponding augmented task graph $\tilde{T}_{GJ,m}$ or $\tilde{T}_{GS,m}$ is constructed by adding arcs from (k,i) to $(k+1,i)$ for all $k \in \{1,...,m-1\}, i \in \{1,...,n\}$.*

The added constraints produce an additional simplification in the implementation of the parallel algorithm, because the constraints prevent any subcircuit from being active in more than one iteration at any moment of time. For example, tasks (1, 1) and (2, 1) are not allowed to execute concurrently, because these are two instances of the same subcircuit in different iterations. However, different subcircuits are allowed to be active in different iterations concurrently, to the extent permitted by the original task graph. For example, in the task graph of Fig. 2.2(a), tasks (1, 4) and (2, 1) may be executed concurrently, because there is no path connecting them. The fact that only one instance of each subcircuit can be active at a time means that some data structures can be allocated once for each subcircuit and used by all instances without conflict.

Adding arcs to the task graph can reduce the degree of parallelism by requiring some tasks to be executed serially, which previously could have been executed in parallel. The degree by which the parallelism is reduced by the additional arcs is investigated in the following theorems.

Definition 2.7. *For a given G, let $\beta_{GJ}(s)$ and $\beta_{GS}(s)$ be the maximum number of instances of subcircuit s that can be active concurrently using the full window technique based on the unaugmented task graphs $T_{GJ,\infty}$ and $T_{GS,\infty}$, respectively, where the task execution times and number of processors are arbitrary.*

Definition 2.8. *For any directed graph H in which each arc is designated as a feedback or feedforward arc, let $f(H)$ be the number of feedforward arcs in H and let $b(H)$ be the number of feedback arcs in H.*

Theorem 2.1. *If s is a vertex of G then*

$$\beta_{GJ}(s) = \begin{cases} f(C_1)+b(C_1) & \text{, if } s \text{ is in a cycle} \\ \infty & \text{, otherwise} \end{cases} \tag{2.8}$$

and

$$\beta_{GS}(s) = \begin{cases} b(C_2) & \text{, if } s \text{ is in a cycle} \\ \infty & \text{, otherwise} \end{cases} \tag{2.9}$$

where C_1 is a cycle of G which minimizes $f(C_1)+b(C_1)$ *and C_2 is a cycle of G* which minimizes $b(C_2)$.

To prove the theorem, first consider the Gauss-Jacobi case and suppose s is in a cycle. For each $k>0$ there is a path from (k,s) to $(k+f(C_1)+b(C_1),s)$ in $T_{GJ,\infty}$, corresponding to one traversal of C_1. Therefore, (k,s) and $(k+i[f(C_1)+b(C_1)],s)$ cannot be active simultaneously, for any $i \in \{1,2,\cdots\}$. If $(k_1,s), \ldots, (k_\gamma,s)$ are active simultaneously, then $k_1 \, mod \, (f(C_1)+b(C_1)), \ldots, k_\gamma \, mod \, (f(C_1)+b(C_1))$ must be distinct. Hence $\gamma \leqslant f(C_1)+b(C_1)$ which implies

$$\beta_{GJ}(s) \leqslant f(C_1)+b(C_1). \tag{2.10}$$

Note that the first $f(C_1)+b(C_1)$ instances of s are mutually independent, since any directed path connecting two of them would violate the fact that C_1 minimizes $f(C_1)+b(C_1)$. If these independent tasks have arbitrarily large computation times compared to all other tasks, they will be active concurrently, given enough processors. Therefore,

$$\beta_{GJ}(s) \geq f(C_1)+b(C_1). \tag{2.11}$$

Combining (2.10) and (2.11) verifies the first part of (2.8). If s is not in a cycle, all instances of s are independent, and an arbitrary number of them can be active concurrently, thus completing the proof of (2.8). The Gauss-Seidel result is obtained by noting that for each $k > 0$ there is a path from (k, s) to $(k + b(C_2), s)$ in $T_{GS,\infty}$, corresponding to one traversal of C_2, and then following the same argument as for the Gauss-Jacobi case to complete the proof.

In circuit simulations where a high degree of accuracy is required, accurate models are used which invariably have bidirectional coupling between each pair of nodes that are connected through a circuit element. Even MOS transistor gate terminals, which are nearly unidirectional, are subject to capacitive feedback from the source and drain terminals. In terms of the subcircuit graph, a circuit which has only bidirectional coupling will have an arc from vertex j to vertex i for each arc from i to j. The number of possible concurrent instances of subcircuits is severely restricted for circuits with bidirectional coupling, as evidenced by the next theorem.

Theorem 2.2. *If all coupling in G is bidirectional, and if G contains no isolated vertices, then* $\beta_{GJ}(s)=2$ *and* $\beta_{GS}(s) = 1$.

If s is a vertex of G, then there exists a vertex j such that there are arcs from s to j and from j to s. These two arcs constitute a cycle C containing vertex s, such that $f(C)=b(C)=1$. Since any cycle must contain at least one feedforward arc and one feedback arc, C minimizes the expressions $f(C)+b(C)$ and $b(C)$. The proof is completed by applying Theorem 2.1.

The preceding theorem indicates that the extra arcs of the augmented graph have absolutely no effect on the parallelism of the Gauss-Seidel method for bidirectionally coupled circuits. But in the Gauss-Jacobi case, a factor of up to 2 may be sacrificed in parallelism. In the definition of β_{GJ}, the computation times of the tasks are unspecified, and consequently, β_{GJ} represents the worst possible combination of task times. It is readily apparent that no concurrently active instances of the same subcircuit are possible in parallel Gauss-Jacobi if all the tasks in the task graph require the same amount of computation time and if the number of processors is greater than or equal to the number of subcircuits. In this case all tasks in iteration k are active in time interval k, and no overlapping of iterations occurs. In this case the extra arcs of the augmented graph have no practical effect on the parallelism. This observation motivates the following theorem which relates the loss of Gauss-Jacobi parallelism in the $\tilde{T}$ augmented graph to the degree of mismatch in the task sizes. First the concepts of task graph depth and width are generalized for graphs in which the vertices have weights representing the computation times of the tasks.

Definition 2.9. *Let $D_w(T)$ and $W_w(T)$ represent the weighted depth and weighted average width of directed graph T with weighted vertices, where $D_w(T)$ is the sum of the vertex weights in a path which has a maximum vertex weight sum, and $W_w(T)$ is equal to the sum of all vertex weights divided by $D_w(T)$.*

Then D_w is the parallel completion time and W_w is the average parallelism, or the average number of active processors, assuming an unlimited supply of processors.

Theorem 2.3. *If all of the following conditions hold,*

(a) *G has only bidirectional coupling and no isolated vertices;*

(b) *each instance of subcircuit i requires computation time w_i, for each $i \in \{1,2,...,n\}$;*

(c) *j is the subcircuit with the maximum computation time $w_j = w_{\max}$; and*

(d) *k is the subcircuit adjacent to j such that no other subcircuit adjacent to j has a larger computation time;*

then

$$\frac{D_w(\tilde{T}_{GJ,m})}{D_w(T_{GJ,m})} \leqslant \frac{2w_{\max}}{w_{\max}+w_k}. \tag{2.12}$$

Each longest path in $T_{GJ,m}$ and each path of maximum weight in $\tilde{T}_{GJ,m}$ has exactly m vertices, one in each iteration. Clearly the path with the largest weight in the augmented graph contains all the instances of the subcircuit with the largest weight. Hence

$$D_w(\tilde{T}_{GJ,m}) = mw_{\max}. \tag{2.13}$$

In the unaugmented graph, there exists a path starting at $(1, j)$ which alternates between instances of subcircuits j and k. Consequently,

$$D_w(T_{GJ,m}) \geqslant \left\lceil \frac{m}{2} \right\rceil w_{\max} + \left\lfloor \frac{m}{2} \right\rfloor w_k. \tag{2.14}$$

Since $w_{\max} \geqslant w_k$, it follows that

$$D_w(T_{GJ,m}) \geqslant \frac{m}{2}(w_{\max}+w_k). \tag{2.15}$$

Combining (2.13) and (2.15) produces the inequality in (2.12), completing the proof.

Theorem 2.3 implies that if the largest subcircuit is adjacent to a subcircuit of nearly the same size, then the parallel completion time will not be significantly affected by the augmented arcs of the task graph. However, if the largest subcircuit is adjacent only to subcircuits which are much smaller, then the loss of parallelism will approach the worst case factor of 2.

2.6 Conclusion

Four parallel algorithms for waveform relaxation have been presented and analyzed using a simplified computation model. The parallel algorithms are derived by using Gauss-Seidel and Gauss-Jacobi relaxation in combination with the full window technique or the time point pipelining technique for exploiting parallelism. The choice between Gauss-Jacobi and Gauss-Seidel represents a tradeoff between greater parallelism and faster convergence, and the choice between time point pipelining and the full window technique represents a tradeoff between greater parallelism and lower overhead. Task graph models for the algorithms have been defined which serve as the basis for analysis and implementation of the algorithms in subsequent chapters.

CHAPTER 3

GAUSS-JACOBI/GAUSS-SEIDEL COMPARISON THEORY

The properties of the Gauss-Jacobi and Gauss-Seidel methods for waveform relaxation are closely related to the more basic Gauss-Jacobi and Gauss-Seidel algorithms for the solution of systems of linear algebraic equations, which are examined in this chapter. The algorithms are compared in terms of convergence speed and parallelism. For the systems of equations which arise at each time point in the solution of node equations of MOS circuits, when the time step is sufficiently small, it will be shown that parallel Gauss-Jacobi is asymptotically faster than parallel Gauss-Seidel, when a sufficiently large number of processors is used. The theorem which establishes this result relates the spectral radii of the iteration matrices to the available parallelism of the methods. A formula is also derived which compares the parallelism of the two methods in terms of the structural properties of the equations being solved.

3.1 Gauss-Jacobi and Gauss-Seidel Relaxation

Consider the problem of solving $Ax=b$ by relaxation, where $x, b \in \mathbb{R}^n$, $A \in \mathbb{R}^{n \times n}$ is nonsingular, and the diagonal elements of A are nonzero. The i^{th} equation is solved independently for x_i while using previously computed or guessed values for the other variables. The update equation for the i^{th} vector element on the k^{th} iteration for Gauss-Jacobi is given by

$$x_i^{(k)} = \frac{1}{a_{i,i}} \left[b_i - \sum_{j=1}^{i-1} a_{i,j} x_j^{(k-1)} - \sum_{j=i+1}^{n} a_{i,j} x_j^{(k-1)} \right] \qquad (3.1)$$

and for Gauss-Seidel by

$$x_i^{(k)} = \frac{1}{a_{i,i}} \left[b_i - \sum_{j=1}^{i-1} a_{i,j} x_j^{(k)} - \sum_{j=i+1}^{n} a_{i,j} x_j^{(k-1)} \right]. \tag{3.2}$$

Let D, L, and U be diagonal, strictly lower triangular, and strictly upper triangular matrices, respectively, such that $A = L + D + U$. Then the Gauss-Jacobi and Gauss-Seidel iteration matrices are given by $M_{GJ} = -D^{-1}(L+U)$ and $M_{GS} = -(L+D)^{-1}U$, respectively. The asymptotic convergence rates are related to $\rho(M_{GJ})$ and $\rho(M_{GS})$, where ρ denotes spectral radius, since these are the factors by which the errors are reduced on each iteration for general initial guesses, as the iteration count approaches infinity.

The Stein-Rosenberg theorem [Var62] relates $\rho(M_{GJ})$ and $\rho(M_{GS})$, and consequently the convergence speeds of Gauss-Jacobi and Gauss-Seidel, for a class of matrices.

Theorem 3.1. *Stein-Rosenberg: If M_{GJ} is nonnegative and $\rho(M_{GJ}) < 1$, then $\rho(M_{GS}) \leqslant \rho(M_{GJ})$.*

The condition that M_{GJ} is nonnegative is equivalent to requiring that $a_{i,j}/a_{i,i} \leqslant 0$ for all $i \neq j$. The condition $\rho(M_{GJ}) < 1$ is necessary and sufficient to assure convergence of Gauss-Jacobi. Matrices arising in the transient analysis of MOS circuits satisfy both these conditions when the time step is sufficiently small, provided that a capacitor is present from each node to ground, and the gate-drain and gate-source capacitances are included in each MOS transistor [Whi86]. For these matrices, the Stein-Rosenberg theorem implies that for a sufficiently small error tolerance, Gauss-Seidel will generally converge in fewer iterations than Gauss-Jacobi. However, on parallel processors, it is possible to perform m iterations of Gauss-Jacobi in less time than m iterations of Gauss-Seidel. Therefore, Theorem 3.1 does not indicate which method will be faster on parallel processors.

3.2 Parallel Gauss-Jacobi and Gauss-Seidel

The unknown update equations for Gauss-Jacobi and Gauss-Seidel, (3.1) and (3.2), involve the same computation; in each case the i^{th} unknown is updated by summing $n-1$ products and a constant. Parallelism can be exploited in computing the products and performing the summation, but the parallelism of these computations will be the same for both Gauss-Jacobi and Gauss-Seidel. The difference between Gauss-Jacobi and Gauss-Seidel that affects how much total parallelism can be exploited is that in the case of Gauss-Jacobi all the unknowns can be updated simultaneously on each iteration, whereas in the case of Gauss-Seidel the number of unknowns which can be updated simultaneously is limited by data dependencies between different unknowns of the same iteration.

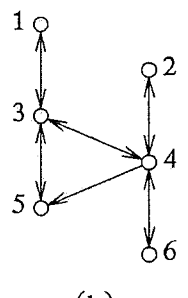
In order to examine the difference between the two methods, let (3.1) and (3.2) be treated as atomic operations which can be computed in one processor step. Using this convention, and assuming that at least n processors are available, one iteration of Gauss-Jacobi takes one processor step, as all the unknown updates can be performed simultaneously, and one iteration of Gauss-Seidel takes n processor steps if A is full, as the i^{th} unknown must be updated before the $i+1$ update equation can be completed.

When A is sparse, it is possible to exploit additional parallelism in Gauss-Seidel to reduce the number of steps required for one iteration to well below n. The sparsity of A allows some updates of a given iteration to be done simultaneously. For example, if $a_{i+1,i}=0$, then $x_i^{(k)}$ can be updated simultaneously with $x_{i+1}^{(k)}$. It is also possible to begin iteration $k+1$ before completing iteration k. For example, if $a_{1,j}$ through $a_{1,n}$ are all zero, then one can compute $x_1^{(k+1)}$ without waiting for $x_j^{(k)}$ through $x_n^{(k)}$ to be computed first.

The number of steps required to compute m Gauss-Seidel or Gauss-Jacobi iterations on a sparse matrix can be determined from an m-iteration task graph, $T_{GS,m}$ or $T_{GJ,m}$, respectively. Each vertex represents the task of performing an update as specified by (3.1) or (3.2), and each arc represents a data dependency. The graph can be constructed based on the nonzero structure of A. First the nonzero structure of A is represented by a directed graph G which is analogous to the subcircuit graph defined previously. The vertices of G are numbered 1 to n, corresponding to the vector element numbers, and each off-diagonal element $a_{j,i}$ is represented by an arc from i to j. For example, Fig. 3.1 shows the nonzero structure of a matrix and its corresponding graph G. The task graphs $T_{GJ,m}$ and $T_{GS,m}$ can be constructed based on G in the same

$$\begin{bmatrix} \times & 0 & \times & 0 & 0 & 0 \\ 0 & \times & 0 & \times & 0 & 0 \\ \times & 0 & \times & \times & \times & 0 \\ 0 & \times & \times & \times & 0 & \times \\ 0 & 0 & \times & \times & \times & 0 \\ 0 & 0 & 0 & \times & 0 & \times \end{bmatrix}$$

(a)



(b)

Figure 3.1. (a) Nonzero structure of A ; (b) the corresponding graph G .

manner as described in Chapter 2.

3.3 The Parallelism Ratio

A comparison of the times required for the parallel Gauss-Jacobi and parallel Gauss-Seidel methods must address their relative degrees of parallelism. The parallelism ratio specified in the following definition will be used in comparing the asymptotic convergence behavior of the methods as the number of iterations goes to infinity.

Definition 3.1. *Let the parallelism ratio r be defined by the equation*

$$r = \lim_{m \to \infty} \frac{D(T_{GS,m})}{D(T_{GJ,m})}. \tag{3.3}$$

The existence of the limit and the relationship of r to the structure of A will be treated in a subsequent section of this chapter. Equation (3.3) implies that m iterations of Gauss-Seidel require r times as many processor steps as m iterations of Gauss-Jacobi, in the limit as the iteration count goes to infinity. The ratio r is directly related to the number of Gauss-Seidel and Gauss-Jacobi iterations that can be performed in a given number of processor steps, and this relationship is given in Lemma 3.4 following a definition and several other lemmas which establish some basic properties of the task graph depths. First, functions are defined which represent the maximum number of iterations that can be performed in a given number of processor steps.

Definition 3.2. *Define the functions $m_{GS}, m_{GJ} : \mathbb{Z} \to \mathbb{Z}$ such that $m_{GS}(l)$ is the largest integer satisfying $D(T_{GS,m_{GS}(l)}) \leqslant l$, and $m_{GJ}(l)$ is the largest integer satisfying $D(T_{GJ,m_{GJ}(l)}) \leqslant l$.*

In the degenerate case in which G contains no cycles, the exact solution is obtained in a finite number of iterations, as reflected in the following lemma.

Lemma 3.1. *If G does not contain a cycle, and $M \geqslant 0$ is the length of a longest path in G, then*

$$D(T_{GJ}, m) = D(T_{GS}, m) = M+1, \text{ for all } m \geqslant M+1. \tag{3.4}$$

If $M=0$, G contains no arcs, and the task graphs contain no arcs. Therefore, the task graph depths are 1. If G contains a longest path P of length $M>0$, then there exist corresponding paths P_{GJ} in $T_{GJ,\infty}$ and P_{GS} in $T_{GS,\infty}$ both of length M. If either $T_{GJ,\infty}$ or $T_{GS,\infty}$ contains a longer path, then there is a corresponding path in G which contradicts the fact that P is a longest path. Therefore, $D(T_{GJ,\infty}) = D(T_{GS,\infty}) = M+1$. Since adjacent vertices of a task graph are always in the same or consecutive iterations, any task graph for at least $M+1$ iterations will contain the longest path of length M, and have depth $M+1$.

In the more interesting case in which G contains a cycle, the Gauss-Jacobi iterations proceed at a rate of exactly one iteration per processor step.

Lemma 3.2. *If G contains a cycle, then $D(T_{GJ,m}) = m$, and $m_{GJ}(l) = l$, for all $m, l \in \mathbb{Z}$.*

Due to the construction of $T_{GJ,m}$, the vertices in any path must have consecutive iteration numbers. Therefore, $D(T_{GJ,m}) \leqslant m$. Since G contains a cycle, it contains a walk of length $m-1$, and such a walk corresponds to a path of length $m-1$ in $T_{GJ,m}$. Therefore, $D(T_{GJ,m}) \geqslant m$. Hence, $D(T_{GJ,m}) = m$. The relationship $m_{GJ}(l) = l$ follows immediately, using the definition of m_{GJ}.

The Gauss-Seidel task graph depth is more difficult to characterize. In fact the expression $D(T_{GS,m+1}) - D(T_{GS,m})$ is not necessarily constant as a function of m. Some useful, but relatively weak, properties of the Gauss-Seidel task graph depths are given in the following lemma.

Lemma 3.3. *If G contains a cycle, then the sequences $D(T_{GS,m})$ and $m_{GS}(l)$ are unbounded and monotone nondecreasing in m and l, respectively.*

Since G contains a cycle, it contains a walk of infinite length which maps to a path of infinite length in $T_{GS,\infty}$. Consequently, the task graph depth goes to infinity. The addition of one more iteration to any task graph of finite iterations does not remove any arcs or vertices from the original graph and therefore does not reduce the depth. Hence, the depth is monotone nondecreasing. As a result, $m_{GS}(l)$ must also be unbounded and monotone nondecreasing.

The ratio r can now be related to the number of iterations which can be performed in a given number of processor steps. The following lemma indicates that the Gauss-Jacobi method can perform r times as many iterations as Gauss-Seidel, in a given number of processor steps, in the limit as the number of processor steps goes to infinity, for the nondegenerate case where G contains a cycle.

Lemma 3.4. *If G contains a cycle, then*

$$\lim_{l\to\infty} \frac{m_{GJ}(l)}{m_{GS}(l)} = r. \tag{3.5}$$

Due to Lemma 3.2 and Def. 3.1, for any $\epsilon>0$ there exists $M \in \mathbb{Z}$ such that

$$\left| \frac{D(T_{GS,m})}{m} - r \right| < \epsilon, \text{ for all } m > M, \tag{3.6}$$

which implies

$$D(T_{GS,m}) = [\delta(m)+r]m, \text{ for all } m > M, \tag{3.7}$$

where $|\delta(m)|<\epsilon$. Since $m_{GS}(l)$ is unbounded and nondecreasing, there exists $L \in \mathbb{Z}$ such that $m_{GS}(l)>M$, for all $l>L$. The definition of m_{GS} and (3.7) imply that for $l>L$, $m_{GS}(l)$ is given by the largest integer m satisfying

$$[\delta(m)+r]m \leqslant l. \tag{3.8}$$

Consequently, for any $\epsilon > 0$ there exists L such that

$$m_{GS}(l) = \left\lfloor \frac{l}{\delta(m_{GS}(l))+r} \right\rfloor, \text{ for all } l > L, \tag{3.9}$$

where $|\delta(m_{GS}(l))| < \epsilon$. Dividing the equality $m_{GJ}(l)=l$ by (3.9) yields

$$\frac{m_{GJ}(l)}{m_{GS}(l)} = \frac{l}{\left\lfloor \frac{l}{\delta(m_{GS}(l))+r} \right\rfloor}, \text{ for all } l > L. \tag{3.10}$$

Since $\delta(m_{GS}(l))$ can be made arbitrarily small by making L large, (3.5) must hold, and the proof of the lemma is complete.

3.4 Parallel Convergence Speed

If m iterations of Gauss-Seidel can be completed in l processor steps, then rm iterations of Gauss-Jacobi can be completed in the same number of processor steps, in the limit as l goes to infinity. In the limit, the error is multiplied by a factor of $\rho(M_{GS})^m$ in m iterations of Gauss-Seidel, and it is multiplied by a factor of $\rho(M_{GJ})^{rm}$ by Gauss-Jacobi in an equal number of processor steps. Therefore, Gauss-Jacobi will be asymptotically faster than Gauss-Seidel if $\rho(M_{GJ})^r \leqslant \rho(M_{GS})$. In the following theorem, which is the main result of this chapter, it will be shown that this relationship between the spectral radii holds for the class of matrices to which the Stein-Rosenberg theorem applies. Therefore, parallel Gauss-Jacobi is asymptotically faster than parallel Gauss-Seidel for these matrices [Sma88a].

Theorem 3.2. *If M_{GJ} is nonnegative and $\rho(M_{GJ})<1$, then $\rho(M_{GJ})^r \leqslant \rho(M_{GS})$.*

If $\rho(M_{GJ})=0$ then $\rho(M_{GS})=0$, and the theorem is trivially satisfied [Var62]. For the case where $\rho(M_{GJ})>0$, G must contain a cycle because the iterations do not converge in a finite number of iterations. For this case, the proof of Theorem 3.2 utilizes the following lemma [Gan59] and definition:

Lemma 3.5. *Perron-Frobenius: If matrix $M \in \mathbb{R}^{n \times n}$ is nonnegative, then it has a nonnegative real eigenvalue equal to its spectral radius and a nonnegative eigenvector associated with that eigenvalue.*

Definition 3.3. *Let $y^{(l)}, z^{(l)} \in \mathbb{R}^n$ contain the most recently computed iterate values for each vector element after l processor steps of the parallel Gauss-Jacobi and Gauss-Seidel algorithms respectively.*

Since a Gauss-Jacobi iteration finishes in one processor step, $y^{(l)}$ is equal to the l^{th} Gauss-Jacobi iterate, and therefore

$$y_i^{(l)} = \frac{b_i}{a_{i,i}} + \sum_{j \neq i, j=1}^{n} -\frac{a_{i,j}}{a_{i,i}} y_j^{(l-1)}. \tag{3.11}$$

In general, $z^{(l)}$ never corresponds to any iterate of Gauss-Seidel, because overlapping of the iterations implies that the most recently computed element values can correspond to different iterations. And because of data dependencies inherent in the Gauss-Seidel algorithm, many of the elements of $z^{(l)}$ are the same as those of $z^{(l-1)}$. Therefore, each Gauss-Seidel update will be given by either

$$z_i^{(l)} = z_i^{(l-1)} \tag{3.12a}$$

or

$$z_i^{(l)} = \frac{b_i}{a_{i,i}} + \sum_{j \neq i, j=1}^{n} -\frac{a_{i,j}}{a_{i,i}} z_j^{(m(l,i,j))}, \tag{3.12b}$$

where $m(l,i,j) \in \{0, \cdots, l-1\}$. Note that $m(l,i,j)$ is used to indicate that in order to follow the Gauss-Seidel update formula, it may be necessary to pick out elements from several different, but earlier, z vectors.

To complete the proof, consider the problem of solving $Ax=0$ by relaxation, where A is such that M_{GJ} exists and is nonnegative and $0<\rho(M_{GJ})<1$. Let the initial guess $x^{(0)}=y^{(0)}=z^{(0)}$ be a nonnegative eigenvector associated with a nonnegative eigenvalue of M_{GJ} equal to $\rho(M_{GJ})$. Since M_{GJ} is nonnegative, $a_{i,j}/a_{i,i} \leqslant 0$ for all $i \neq j$. Consequently, since $x^{(0)}$ is nonnegative and $b=0$, each term in (3.11) and (3.12b) is nonnegative for all i, l. Also, since $y^{(0)}$ is an eigenvector associated with an eigenvalue equal to $\rho(M_{GJ})<1$, $y^{(l)}=\rho(M_{GJ})^l y^{(0)}$, and therefore $y_i^{(l)}$ decays monotonically with l for all i.

It will be shown by induction that

$$y_i^{(l)} \leqslant z_i^{(m)}, \text{ for all } i, \text{ for all } m \in \{0, \cdots, l\}, \tag{3.13}$$

holds for all l. Clearly (3.13) holds for $l=0$, forming the basis of the induction. Assume that (3.13) holds for a given l, and consider processor step $l+1$. In those cases where (3.12a) applies, $y_i^{(l)} \leqslant z_i^{(l)} = z_i^{(l+1)}$, and $y_i^{(l+1)} \leqslant z_i^{(l+1)}$ because $y_i^{(l)}$ is monotone decreasing in l. In those cases where (3.12b) applies, each term of the summation in (3.11) is less than or equal to the corresponding term in (3.12b) and all the terms are nonnegative. Hence $y_i^{(l+1)} \leqslant z_i^{(l+1)}$. Since $y_i^{(l)}$ decreases monotonically, $y_i^{(l+1)} \leqslant y_i^{(m)} \leqslant z_i^{(m)}$ for all $m \leqslant l$. Consequently, $y_i^{(l+1)} \leqslant z_i^{(m)}$ for all i and all $m \in \{0, \cdots, l+1\}$, thus completing the induction.

In l processor steps, $m_{GS}(l)$ iterations of Gauss-Seidel are completed. Let $x^{(m_{GS}(l))}$ be the Gauss-Seidel iterate on iteration $m_{GS}(l)$. Then for any i, l, there exists $m \leqslant l$ such that $x_i^{(m_{GS}(l))} = z_i^{(m)}$. Applying (3.13) for each i, it follows that $y^{(l)} \leqslant x^{(m_{GS}(l))}$, where $y^{(l)}$ is equal to the l^{th} Gauss-Jacobi iterate. As $l \to \infty$, Lemma 3.4 indicates that r times as many Gauss-Jacobi iterations are completed as Gauss-Seidel iterations. In order for $x^{(m_{GS}(l))}$ to remain larger than $y^{(l)}$, M_{GS} must have an eigenvalue with magnitude larger than $\rho(M_{GJ})^r$, the factor by which the Gauss-Jacobi error is reduced in r processor steps. Therefore, $\rho(M_{GJ})^r \leqslant \rho(M_{GS})$, and the proof is complete.

3.5 Parallelism Ratio Bounds

It is interesting to note that the proof of Theorem 3.2 does not require explicit knowledge of the value of r, but rather only assumes that r exists. The value of r is of interest because it represents the degree of parallelism of Gauss-Jacobi compared to Gauss-Seidel. On the average, parallel Gauss-Jacobi requires r times as many processors as parallel Gauss-Seidel if the full parallelism of the methods is employed. In this section, several bounds on r will be presented, assuming that r exists. In the next section, the existence of r will be established, and an exact formula for r will be given in terms of properties of G.

In the usual case where G contains a cycle, and m Gauss-Jacobi iterations require exactly m processor steps, r is equivalent to the average number of steps between the completion of successive Gauss-Seidel iterations. If no parallelism is employed in Gauss-Seidel, n extra steps are required for each extra iteration. If parallelism is utilized, the number of steps per iteration will be no greater than n, and therefore

$$r \leqslant n. \tag{3.14}$$

This bound will be reached if A is full, or if G contains a cycle from vertex 1 to 2 to ... to n to 1.

If concurrent updates are allowed within a Gauss-Seidel iteration, then $D(T_{GS,1})$ steps will be required for the first iteration. And if parallelism is not exploited between different iterations, each additional iteration will require an additional $D(T_{GS,1})$ steps. Since some, but not all, of the potential parallelism is utilized, it follows that

$$r \leqslant D(T_{GS,1}) \leqslant n. \tag{3.15}$$

This tighter bound on r will be approached when nearly all of the Gauss-Seidel parallelism is due to intra-iteration parallelism.

In some cases, most or all of the available Gauss-Seidel parallelism arises from the overlapping of iterations. For example, if A is tridiagonal then $D(T_{GS,1})$ is equal to n, since each update requires the result of the previous update of the same iteration. However, iteration $k+1$ can begin after only 2 steps of iteration k. Because of the inter-iteration parallelism, $r=2$ for a tridiagonal matrix, regardless of the value of n. In cases like this, the bounds in (3.15) do not give an accurate indication of the high degree of Gauss-Seidel parallelism. An exact determination of the value of r requires that both inter-iteration and intra-iteration parallelism be taken into account.

3.6 Parallelism Ratio Formula

In this section, the limit in (3.3) which defines r will be shown to exist, and a formula will be derived for r in terms of properties of G. In the following development, $L(P)$ will denote the length of path P, and $D(v)$ will denote the depth of vertex v, defined as the number of vertices in a longest path terminating at v. The notation $D(T)$, where T is a directed graph, will denote the depth of the graph as defined in Chapter 2. The main result of this section is given in Theorem 3.3.

Theorem 3.3. *The limit in (3.3) exists, and*

$$r = \begin{cases} 1+f(C)/b(C) & \text{, if } G \text{ contains a cycle} \\ 1 & \text{, otherwise} \end{cases} , \tag{3.16}$$

where C is a cycle of G which maximizes $f(C)/b(C)$.

Note that any cycle of G must contain a feedback arc, so $b(C)$ will be nonzero in (3.16) when G contains a cycle.

If G contains no cycles, then $D(T_{GJ,m})=D(T_{GS,m})$ for m sufficiently large, due to Lemma 3.1. Therefore, $r=1$ in this case.

If G contains a cycle, then both the Gauss-Seidel and Gauss-Jacobi task graph depths go to infinity, and $D(T_{GJ,m})=m$. To complete the proof, it must be shown that $\lim[D(T_{GS,m})/m]=1+f(C)/b(C)$, when G contains a cycle. This will be done in the next two lemmas by establishing upper and lower bounds on the limit.

Lemma 3.6. *If G contains a cycle, then for any $\epsilon>0$ there exists $M \in \mathbb{Z}$ depending on ϵ such that*

$$\frac{D(T_{GS,m})}{m} > 1 + \frac{f(C)}{b(C)} - \epsilon, \text{ for all } m > M, \tag{3.17}$$

where C is a cycle of G which maximizes $f(C)/b(C)$.

Let v be a vertex of C. For any $j \geqslant 1$, let $m=jb(C)+1$. Let v_1 and v_m be the instances of v in iterations 1 and m of $T_{GS,m}$, respectively. There is a path P from v_1 to v_m corresponding to j traversals of C, because the iteration number increases by one every time a feedback arc is traversed. Combining the relations $D(T_{GS,m})\geqslant D(v_m)$, $D(v_m)\geqslant D(v_1)+L(P)$, $D(v_1)\geqslant 1$, $L(P)=j[f(C)+b(C)]$, and $j=(m-1)/b(C)$ produces

$$D(T_{GS,m}) \geqslant 1+\frac{m-1}{b(C)}[f(C)+b(C)], \text{ for } m \in \{1, b(C)+1, 2b(C)+1, \cdots\}. \tag{3.18}$$

Bounds on the task graph depth for intermediate values of m can be obtained using the fact that $D(T_{GS,m})$ is monotone nondecreasing in m. The following lower bound on $D(T_{GS,m})$ applies for all m:

$$D(T_{GS,m}) \geqslant 1+\frac{m-b(C)}{b(C)}[f(C)+b(C)]. \tag{3.19}$$

Dividing by m gives

$$\frac{D(T_{GS,m})}{m} \geqslant \frac{1}{m}+\left[\frac{1}{b(C)}-\frac{1}{m}\right]\left[f(C)+b(C)\right]. \tag{3.20}$$

Discarding the $1/m$ term and distributing yields

$$\frac{D(T_{GS,m})}{m} \geqslant 1+\frac{f(C)}{b(C)}-\frac{f(C)+b(C)}{m}. \tag{3.21}$$

Note that $f(C)+b(C)<n$, as C contains at most $n-1$ arcs. Therefore,

$$\frac{D(T_{GS,m})}{m} > 1+\frac{f(C)}{b(C)}-\frac{n}{m}. \tag{3.22}$$

By setting $M=n/\epsilon$, (3.17) is obtained, thus completing the proof of Lemma 3.6.

Lemma 3.7. *If G contains a cycle, then for any $\epsilon>0$ there exists $M \in \mathbb{Z}$ depending on ϵ such that*

$$\frac{D(T_{GS,m})}{m} < 1+\frac{f(C)}{b(C)}+\epsilon, \text{ for all } m>M, \tag{3.23}$$

where C is a cycle of G which maximizes $f(C)/b(C)$.

Let u be any vertex of G and let u_m be the instance of u in iteration m of $T_{GS,m}$. Let P be any longest path, consisting of one or more vertices, terminating at u_m. In order to establish a bound on the length of P, partition P into subpaths $P_0, Q_1, P_2, Q_2, \ldots,$ P_{s-1}, Q_s, P_s, such that the following conditions are satisfied:

(a) The concatenation of the arcs of $P_0, Q_1, \cdots, P_s$ is equal to the sequence of arcs in P.

(b) P_j contains no two instances of a common vertex of G, for each j.

(c) The endpoints of Q_j are instances of a common vertex of G, for each j.

(d) The origins of Q_i and Q_j are instances of distinct vertices of G, for each $i \neq j$.

The possibilities that s might be 0 and that P_0 or P_s might be null are not ruled out. Note that conditions (a), (b), and (c) are easily satisfied by putting subpaths with more than one instance of the same vertex into Q subpaths. If a partitioning does not satisfy condition (d), it can be modified by combining Q_i , P_i, ..., Q_j into a single Q_i subpath.

Since P is a longest path terminating at u_m, $D(u_m) = L(P)+1$. Condition (a) then implies

$$D(u_m) = 1 + \sum_{j=0}^{j=s} L(P_j) + \sum_{j=1}^{s} L(Q_j). \tag{3.24}$$

Based on this equation, an upper bound on $D(u_m)$ will be derived which will lead to the result in (3.23). Condition (b) implies that $L(P_j) \leqslant n-1$, and condition (d) implies $s \leqslant n$. Applying these relations to (3.24), plus the fact that $L(Q_j) = f(Q_j) + b(Q_j)$, yields

$$D(u_m) \leqslant 1 + (n+1)(n-1) + \sum_{j=1}^{s} [f(Q_j) + b(Q_j)]. \tag{3.25}$$

Recall that the first and last vertices of Q_j are instances of a single vertex x_j of G. To obtain a bound on $f(Q_j)$, Q_j is further partitioned into one or more subpaths, $Q_{j,1}, \ldots, Q_{j,q}$, such that each subpath starts and ends on an instance of x_j, and no internal vertex is an instance of x_j. Each $Q_{j,i}$ corresponds to a cycle $C_{j,i}$ of G. Because of the manner in which cycle C is chosen, $f(C_{j,i})/b(C_{j,i}) \leqslant f(C)/b(C)$. Using this relationship and summing over all subpaths of Q_j, the bound $f(Q_j) \leqslant b(Q_j) f(C)/b(C)$ is obtained. Substituting this bound into (3.25) produces

$$D(u_m) \leqslant n^2 + \left[1 + \frac{f(C)}{b(C)}\right] \sum_{j=1}^{s} b(Q_j). \tag{3.26}$$

Since (3.26) applies for any choice of vertex u_m in iteration m, $D(u_m)$ may be replaced by $D(T_{GS,m})$. Applying the relationship $\sum b(Q_j) \leqslant b(P) < m$ and dividing by m yields

$$\frac{D(T_{GS,m})}{m} < \frac{n^2}{m} + 1 + \frac{f(C)}{b(C)}. \tag{3.27}$$

Choosing $M = n^2/\epsilon$ confirms the validity of (3.23) and completes the proof of Lemma 3.7. Lemmas 3.6 and 3.7 lead directly to the result stated in Theorem 3.3 for the case where G contains a cycle, thus completing the proof of the theorem.

CHAPTER 4

SPEEDUP ESTIMATES

The objective of parallel processing is to reduce the overall run time of a program by executing different parts of it on different processors concurrently. A convenient measure of the success of parallel processing in a given situation is the *speedup*, which indicates how much faster the program runs on a specified number of processors compared to the run time on a single processor. In this chapter, several techniques are investigated for estimating the speedup of parallel waveform relaxation algorithms for a set of benchmark circuits. The simplifying assumptions introduced in Chapter 2 are used as a starting point for computing simple estimates. The assumptions are then replaced by more realistic assumptions producing more accurate estimates. In the process of refining the assumptions, insights are gained into the extent to which different factors affect the speedup. In subsequent chapters, the speedup estimates are used for two purposes. Fast estimates are used to select the fastest parallel waveform relaxation algorithm prior to performing a circuit simulation. Accurate estimates excluding multiprocessing overhead factors are compared with measured speedups to determine the extent to which overhead affects the performance of the algorithms. Finally, in this chapter, accurate estimates neglecting overhead are used to predict the potential performance of the algorithms when the number of processors is large.

4.1 Speedup

The parallel processing speedup achieved by an algorithm X, applied to a given problem, running on k processors, is defined as

$$S_k = \frac{\tau_1}{\tau_k}, \tag{4.1}$$

where τ_k is the run time of algorithm X on k processors and τ_1 is a reference time which in some sense reflects the time required to solve the same problem using only 1 processor. The speedup is a measure of how much faster the program runs on k processors compared to the time required to obtain the same solution on 1 processor.

One obvious choice for the reference time τ_1 is the time required to run the same algorithm, X, on 1 processor. Speedups computed in this manner will be referred to as unnormalized speedups, denoted as $S_{U,k}$. Unnormalized speedups satisfy the property $S_{U,k} \leqslant k$, and $S_{U,k}$ will be close to k if the computations are nearly evenly distributed between all the processors throughout the execution of the program and if the parallel processing overhead is small. The ratio $S_{U,k}/k$ is a measure of the processor utilization, or the fraction of time, on the average, that the processors are busy, not counting the time spent doing extra work which is performed in the k-processor case but not in the uniprocessor case.

When using speedups to compare the overall performance of different algorithms for solving the same problem, a common reference time must be used for all the algorithms so that a greater speedup indicates an algorithm with a shorter run time. The normalized speedup of algorithm X on k processors with respect to algorithm Y is defined as

$$S_{N,k} = \frac{\tau_{Y,1}}{\tau_{X,k}} = S_{U,k}\,\frac{\tau_{Y,1}}{\tau_{X,1}}, \tag{4.2}$$

where τ_Z, m is the run time of algorithm Z on m processors, for any Z and m. A good choice for the common reference is the time required by the fastest available uniprocessor algorithm. Then $S_{N,k} \leqslant S_{U,k} \leqslant k$. Note that $S_{N,k}/k$ is not a measure of the utilization of the processors as is $S_{U,k}/k$.

Of the waveform relaxation algorithms under consideration, the Gauss-Seidel method consistently runs faster than Gauss-Jacobi on a uniprocessor. And a uniprocessor waveform relaxation program which does not have the extra overhead of the full window technique or time point pipelining will run faster than a program with the extra parallel processing code. Therefore, a uniprocessor program using Gauss-Seidel is a useful reference for computing normalized speedups.

4.2 Benchmark Circuits

The performance of the parallel algorithms is a strong function of the circuit being simulated and the subcircuit partitioning, since the subcircuit interconnection structure determines the structure of the task graphs. Five benchmark circuits, including CMOS and NMOS designs, are used to compare the performance of the parallel algorithms. The circuit characteristics are summarized in Table 4.1. All coupling is bidirectional, so the results from Chapter 2 concerning such circuits are applicable.

Circuits *dvs* and *ben2k* are portions of industrial designs which were extracted from chip layouts. The inclusion of nonzero interconnect resistance on the power and ground busses is responsible for the highly nonuniform subcircuit sizes in *dvs*, since there are a large number of tightly coupled bus nodes which are placed in a single

Table 4.1. Circuit Characteristics

Circuit	FETs	Nodes	Subcircuits	Nodes per Subcircuit			
				min	max	mean	σ
dvs	54	189	27	1	30	7.0	6.5
dpla	116	56	30	1	9	1.9	1.6
scdac	416	150	90	1	7	1.7	1.5
ben2k	805	388	119	1	54	3.3	6.2
digfi	698	378	222	1	10	1.7	1.2

subcircuit. The *ben2k* circuit is a portion of a static RAM, and the transistor connections rather than parasitic resistors account for the large subcircuits in this case. The *dpla* circuit contains only a few interconnect resistors, and the *scdac* and *digfi* circuits contain none. These latter circuits have a comparatively high degree of uniformity of subcircuit sizes, as demonstrated by the low maximum numbers of nodes per subcircuit and by the small standard deviations.

4.3 Presimulation Estimates

Two categories of speedup estimates will be addressed: presimulation and post-simulation estimates. The presimulation estimates can be computed without performing a simulation of the circuit. They provide insights into the nature of parallelism in waveform relaxation, and they can serve as the basis for selecting the algorithm to be used for a given circuit on a given number of processors prior to performing the circuit simulation.

4.3.1 Type 1 estimates: uniform task times

Simplifying assumptions were introduced in Chapter 2, and these assumptions are applied to the benchmark circuits to produce the Type 1 speedup estimates for the full window technique. The assumptions are that each task requires the same amount of time, and an unlimited supply of processors is available. Under these conditions,

$$S_{U,\infty}=W(T_{X,k}) \tag{4.3}$$

where X is either GS or GJ, depending on the algorithm used. The Type 1 estimates for the benchmark circuits are given in Table 4.2 for different numbers of iterations, since the number of iterations required for convergence is not known prior to simulation. The number of iterations is typically around 4. The speedups are unnormalized, because the number of iterations and the window boundaries chosen by the automatic

Table 4.2. Type 1 Speedup Estimates

Circuit	Gauss-Seidel Iterations 1	2	4	8	Gauss-Jacobi Iterations 1, 2, 4, or 8
dvs	2.3	2.3	2.4	2.4	27
dpla	2.3	2.9	3.2	3.5	30
scdac	10.0	10.0	12.0	13.0	90
ben2k	3.8	4.7	5.2	5.6	119
digfi	6.3	7.2	7.7	7.9	222

windowing algorithm will differ by unknown amounts for the two relaxation methods, and these factors determine the ratio of uniprocessor run times.

The results show that the Gauss-Seidel speedups are severely limited by the structure of the task graphs. Since the Gauss-Jacobi speedups are 10 to 20 times greater than the Gauss-Seidel speedups, one would expect Gauss-Jacobi to outperform Gauss-Seidel when the number of processors is large, even if Gauss-Jacobi requires considerably more iterations. It is also interesting to note that the Gauss-Seidel speedups are only weakly dependent on the number of iterations, suggesting that most of the parallelism occurs between tasks of an iteration rather than between tasks of different iterations. The Gauss-Jacobi speedups are equal to the number of subcircuits, and are independent of the number of iterations. Since all the task times are assumed to be uniform and all the circuits are bidirectionally coupled, Theorems 2.2 and 2.3 imply that the Type 1 speedup estimates of the benchmark circuits are independent of whether the unaugmented or the augmented, $\tilde{T}$, task graphs are used.

4.3.2 Type 2 estimates: nonuniform task times

The uniform task time assumption automatically causes the tasks to be executed one level at a time in the levelized task graph. Even if the task times are not uniform, it is possible to enforce a rule which requires all tasks in one level to finish before any tasks in the next level begin executing. This is a relatively simple way to enforce the precedence constraints but does not take advantage of all the available parallelism, because some tasks in level $l+1$ may have all their precedence constraints satisfied before all tasks in level l finish.

In the Type 2 speedup estimates, the tasks are forced to be executed one level at a time, but the task times are allowed to be nonuniform. In this scenario, the time required by level l is determined by the largest task in the level. The speedup is estimated as

$$S_{U,\infty} \approx \frac{\sum\limits_{x \in \{\text{all tasks}\}} w_x}{\sum\limits_{l=1}^{D(T)} \max\limits_{x \in \{\text{tasks in level } l\}} \{w_x\}} . \tag{4.4}$$

where T is the task graph and w_x is an estimate of the computation time for task x. Note that nonuniform task times within a level will invariably have a detrimental effect on the speedup in this model. However, differences in task sizes between different levels may result in larger or smaller speedups. If one level contains many tasks and another contains few tasks, then the overall speedup will benefit if the tasks in the level with many tasks all have long computation times compared to the tasks in the level with few tasks. On the other hand, if the computation time of levels with few tasks dominate, then the speedup will suffer.

An estimate is needed of the computation times of the tasks. Since a task consists of the evaluation of a subcircuit over a time window, and since subcircuits are typically small, the task computation time is dominated by the evaluation of model equations. The time to evaluate the model equations depends on the number of models to be evaluated, their complexity, the number of time points at which the models need to be evaluated, and the operating regions of the circuit elements. The number of time points and the operating regions are difficult to estimate without running a circuit simulation, and therefore these factors are ignored in the Type 2 estimates. However, the number and types of models are known from the subcircuit partitioning. Since the number of circuit elements in a subcircuit is typically proportional to the number of nodes, the number of nodes in a subcircuit is used as a measure of the computation time for each task which is an instance of the subcircuit. Therefore, w_x is set equal to the number of nodes in the subcircuit corresponding to task x. The estimate could be further refined by directly considering the number and types of circuit elements in each subcircuit.

The Type 2 estimates are presented for the benchmark circuits in Table 4.3. The nonuniformity of task sizes has a significant effect compared to the results of the Type 1 estimates. The impact is especially severe in the Gauss-Jacobi case, where the speedup is determined by the fraction of circuit nodes which appear in the largest subcircuit. For example, in the *ben2k* circuit, 1/7.2 of the total circuit nodes are contained in the largest subcircuit, even though there are over 100 subcircuits, and this limits the speedup to 7.2. Type 2 estimates are also insensitive to whether or not the $\tilde{T}$ augmented task graphs are used in the case of bidirectionally coupled circuits.

Table 4.3. Type 2 Speedup Estimates

Circuit	Gauss-Seidel Iterations				Gauss-Jacobi Iterations
	1	2	4	8	1, 2, 4, or 8
dvs	1.5	1.5	1.5	1.5	6.3
dpla	1.6	1.9	2.1	2.1	6.2
scdac	3.4	3.9	4.1	4.3	21.4
ben2k	1.6	1.8	1.8	1.9	7.2
digfi	3.5	3.8	3.9	4.0	37.8

4.3.3 Type 3 estimates: unsynchronized levels

The enforcement of a synchronization between each level, as assumed in the Type 2 estimate, has a negative impact on the speedup. To determine the extent of this impact, the Type 3 speedup estimates retain the nonuniform task time estimates of Type 2, while removing the synchronization of levels. In the Type 3 model, each task is assumed to begin executing immediately after all its predecessors in the task graph are finished. If P is a path in the task graph such that no other path has a larger sum of estimated task times, then the parallel execution time is determined by P, and the Type 3 estimate is given by

$$S_{U,\infty} \approx \frac{\sum\limits_{x \in \{\text{all tasks}\}} w_x}{\sum\limits_{x \in \{\text{tasks in } P\}} w_x}. \tag{4.5}$$

Table 4.4 shows the results of this estimation procedure, assuming the unaugmented task graphs are used. As expected, the speedups are greater than those predicted by the Type 2 estimates. For these examples, the penalty for synchronizing the levels is up to a factor of 2, and in most cases is considerably less than 2. In the Gauss-Jacobi case, the speedup for an even number of iterations is determined by the two adjacent subcircuits whose node sum is largest, because there is a path in the task graph which

Table 4.4. Type 3 Speedup Estimates: Unaugmented Task Graph

Circuit	Gauss-Seidel Iterations				Gauss-Jacobi Iterations	
	1	2	4	8	1	2, 4, or 8
dvs	1.5	1.5	1.5	1.5	6.3	7.7
dpla	1.8	2.0	2.2	2.3	6.2	10.2
scdac	5.4	7.0	8.1	8.8	21.4	25.0
ben2k	1.9	2.3	2.5	2.6	7.2	9.0
digfi	4.5	5.0	5.3	5.4	37.8	54.0

alternates between these 2 subcircuits on alternate iterations. For example, in the *ben2k* circuit, there are two adjacent subcircuits whose node sum is 2/9 of the total circuit nodes, resulting in a speedup of 9.

Type 1 and Type 2 estimates are independent of whether the unaugmented task graphs T or the augmented task graphs $\tilde{T}$ are used, because the levels are synchronized. In the Type 3 estimate, the impact of the extra constraints of the augmented graph $\tilde{T}$ can be observed in the Gauss-Jacobi results of Table 4.5. In this case the Gauss-Jacobi completion time is determined entirely by the largest subcircuit, since the task graph contains a path including each instance of the largest subcircuit. Since all the circuits are bidirectionally coupled, the extra constraints of $\tilde{T}_{GS}$ have no effect compared to the unaugmented task graph, as proved in Chapter 2.

4.3.4 Normalization to Gauss-Seidel

The unnormalized speedup estimates considered above fail to demonstrate whether Gauss-Seidel or Gauss-Jacobi will be faster, because the ratio $\tau_{GS,1}/\tau_{GJ,1}$ is unknown. The benchmark circuits were simulated using both the Gauss-Seidel and Gauss-Jacobi methods on a uniprocessor, to obtain measurements of $\tau_{GS,1}$ and $\tau_{GJ,1}$. The ratios are presented in Table 4.6. For all the benchmark circuits, the ratios are very close to 0.7.

Table 4.5. Type 3 Speedup Estimates: Augmented Task Graph $\tilde{T}_{GJ}$

Circuit	Gauss-Jacobi Iterations 1, 2, 4, or 8
dvs	6.3
dpla	6.2
scdac	21.4
ben2k	7.2
digfi	37.8

Table 4.6. Ratios of Gauss-Seidel to Gauss-Jacobi Uniprocessor Run Times

Circuit	Ratio
dvs	0.7
dpla	0.8
scdac	0.7
ben2k	0.6
digfi	0.7

This suggests that it may be appropriate to use the constant 0.7 as an estimate of $\tau_{GS,1}/\tau_{GJ,1}$ in presimulation normalized speedup estimates. Multiplying the Gauss-Jacobi speedup estimates by 0.7 results in normalized speedups which are still considerably larger than the Gauss-Seidel speedups. In Chapter 5, the normalized speedup estimates are applied to the problem of selecting between the Gauss-Seidel and Gauss-Jacobi methods prior to performing a circuit simulation.

4.3.5 Time point pipelining estimates

The parallel performance of time point pipelining is a strong function of the number of time points in each window and the positions of the time points on the time axis. This information is not available prior to performing the circuit simulation. However, for the purpose of predicting which of the 4 parallel waveform relaxation methods is fastest for a given circuit and a given number of processors, it is not necessary to actually predict the time point pipelining speedup. A technique for selecting the fastest of the 4 methods prior to simulation, based on the speedup estimates for the full window technique, is presented in Chapter 6.

4.4 Post-simulation Estimates

Even the most sophisticated of the presimulation estimates suffers from the following limitations:

(1) Differences in the amount of signal activity in different subcircuits affect the task execution times, but are not considered in the estimates.

(2) The Gauss-Seidel/Gauss-Jacobi normalization factor can only be guessed based on previous experience.

(3) Details on the number and locations of time points, which affect time point pipelining performance, are unknown prior to simulation.

(4) Multiprocessing overhead factors such as task scheduling, data communications, and time spent waiting for access to shared resources are neglected.

All of these limitations except the last can be overcome by performing a simulation of the circuit on a uniprocessor prior to computing the estimate, and using detailed information on which tasks are executed and their individual computation times.

Accurate post-simulation estimates neglecting overhead have two applications. The speedup on k processors can be estimated even if a multiprocessor with k processors is not available. This allows projections to be made of the potential performance of the algorithms on machines which will become available in the future. The other application is that of determining the degree to which multiprocessing overhead is responsible for the performance of the algorithms as measured in actual multiprocessor runs. For example, if an algorithm has a speedup of only 2 on 8 processors, then either the processors are idle most of the time due to a lack of work which can be done concurrently, or the multiprocessing overhead is excessive. If the post-simulation estimate of speedup neglecting overhead is 7, then one would conclude that the problem is one of overhead rather than a lack of available parallelism.

Unlike the presimulation estimation techniques, post-simulation estimates are not suitable to serve as a guide in selecting the fastest algorithm for a given circuit and a given number of processors prior to simulating the circuit on a multiprocessor. Since a simulation of the circuit is required prior to computing the post-simulation estimate, the need for running the simulation on a multiprocessor after computing the estimate is obviated.

4.4.1 PARASITE description

The PARASITE program was developed to compute accurate post-simulation estimates of the parallel execution time of any of the four parallel waveform relaxation algorithms, for a given circuit on a given number of processors, neglecting multiprocessing overhead. The organization of the PARASITE system is depicted in Fig. 4.1. First a circuit simulation is performed on the circuit of interest using a uniprocessor waveform relaxation program which has been modified to produce two special output files for the PARASITE program. The first file contains the subcircuit graph, which PARASITE uses to construct task graphs. The second file contains information on each task that was executed. For the full window technique, this file contains for each subcircuit evaluation task (k, i) the measured computation time of the task, $\tau(k, i)$. For time point

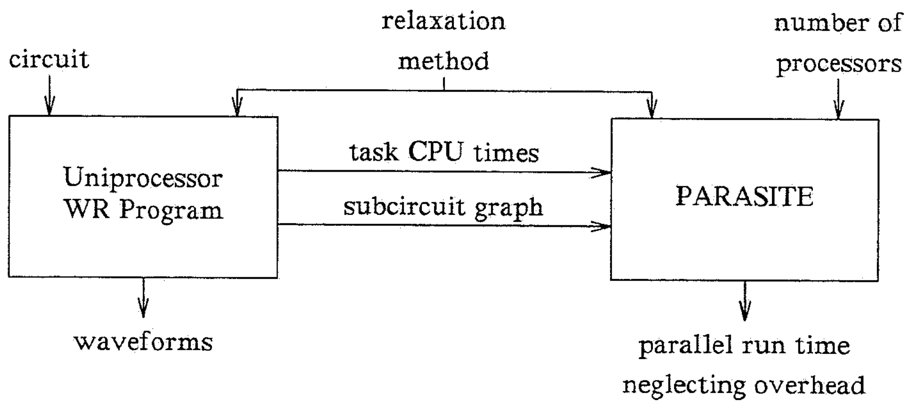


Figure 4.1. PARASITE: Parallel simulation time estimator.

pipelining, the file contains a record of each time point subtask (k, i, t), its measured computation time $\tau(k, i, t)$, and the corresponding initial location of the time point $t_{init}(k, i, t)$.

After the circuit simulation is complete, the PARASITE program is run, also on a uniprocessor. The PARASITE program mimics the operation of the specified parallel waveform relaxation algorithm, but instead of performing computations to solve the circuit equations, it just keeps track of the time that would be required to execute the tasks on a specified number of processors. The windows are processed sequentially. Within each window a weighted task graph T is constructed for either the full window technique or for time point pipelining, as specified. The number of iterations in the task graph is known from the record of tasks which were executed in the circuit simulation. All the other information needed to construct the task graph is contained in the subcircuit graph in the case of the full window technique, and in the subcircuit graph and time point information in the case of time point pipelining. The weights are obtained directly from the measured CPU times of the individual tasks or subtasks.

PARASITE then simulates the parallel execution of the tasks in the graph on N_{procs} processors, using the algorithm given below. In the PARASITE algorithm, t is the estimated elapsed time, t_i is the time at which processor i will finish its current task, x_i is the task assigned to processor i, $w(x)$ is the weight of task x, P is the set of active processors, and $\bar{P}$ is the set of idle processors. A queue is used to hold tasks that are ready to execute but that have not yet been assigned to processors for execution. The function *qput*(*tsk*) puts task *tsk* on the queue, and *qempty*() returns *TRUE* if the queue is empty. The function *qget*() returns a task from the queue and deletes it from the queue. The task obtained by *qget*() is the task of the lowest available iteration number which has been on the queue for the longest time. The initial tasks are those which have an in-degree of 0 in the task graph.

Algorithm 4.1. PARASITE: Simulate Parallel Execution (N_{procs}, T)

```
$P \leftarrow \phi$
$\bar{P} \leftarrow \{1,2,...,N_{procs}\}$
$t, t_1, t_2, \cdots, t_{N_{procs}} \leftarrow 0$
for each (initial task $x$) $qput(x)$
repeat {
                    /* Assign tasks to processors */
    for each ($i \in \bar{P}$) {
        if ($qempty()=FALSE$) {
            $x_i \leftarrow qget()$
            $t_i \leftarrow t + w(x_i)$
            add $i$ to set $P$
            remove $i$ from set $\bar{P}$
        }
    }
                    /* Advance time */
    find $i \in P$ with smallest $t_i$
    $t \leftarrow t_i$
    for each (successor $y$ of $x_i$ in $T$ with no other predecessor) $qput(y)$
    remove $x_i$ from graph $T$
    remove $i$ from set $P$
    add $i$ to set $\bar{P}$
} until ($qempty()=TRUE$ and $P=\phi$)
```

Algorithm 4.1 is used within each window, and the total estimated parallel execution time is the sum of the times in all the windows.

4.4.2 Limitations

The estimates produced by PARASITE are considerably more accurate than the presimulation estimates, but certain limitations should be noted. The fact that PARASITE does not account for parallel processing overhead has been previously discussed, and this feature can be viewed as either an advantage or disadvantage, depending on the intended application. A speedup computed from a PARASITE time estimate is an approximate upper bound on speedup, subject to certain conditions. It is an upper bound because multiprocessing overhead is neglected, thereby resulting in an overestimate of speedup. It is approximate because PARASITE does not compute the optimum schedule of the tasks on the parallel processors. The estimate is subject to certain conditions arising from the assumption that the multiprocessing run executes exactly the same tasks as the uniprocessor case.

PARASITE queues tasks as soon as their precedence constraints are satisfied, and obtains a task from the queue as soon as a processor becomes available to execute a new task. This results in an optimum scheduling of tasks for those cases where the number of processors is either 1 or infinite. For intermediate numbers of processors, the task scheduling may be nonoptimal. When more than one task is on the queue, then the choice of which task to execute next can affect the overall parallel execution time. Even if a multiprocessor waveform relaxation program uses the same policy of obtaining work from the queue on a first-in-first-out basis, small delays introduced by overhead in the multiprocessor run can result in a different order of execution and a different assignment of tasks to processors. Normally the overhead-induced delays will result in a longer run time than the PARASITE estimate, but it is possible for the delays to cause a reduction in the run time, if by chance a more efficient scheduling of tasks on processors results. PARASITE could be modified to determine the optimum schedule of tasks, but this optimization problem is NP-complete and would be too time consuming [Gar79]. In practice, the disruption in execution order caused by overhead delays is unlikely to result in a significantly different schedule, provided the overhead is small compared to the task execution times.

The fact that PARASITE assumes that the same tasks are executed in the uniprocessor and multiprocessor cases has several implications. The study of asynchronous relaxation methods [Cha69, Bau78], in which the task graph evolves during the execution of the relaxation process, is not possible with PARASITE, because the computations performed can depend on the number of processors. The parallel versions of the strict Gauss-Seidel and Gauss-Jacobi relaxation methods do not present a problem for PARASITE because the precedence constraints between tasks are designed to assure that the computations in the parallel case operate on exactly the same data as in the uniprocessor case.

The PARASITE results apply only to the particular circuit, subcircuit partitioning, subcircuit ordering, window boundaries, and relaxation method used in the reference uniprocessor simulation. Each of these factors can affect the uniprocessor run time and the degree of parallelism.

4.4.3 Results

PARASITE was applied to the benchmark circuits, producing the unnormalized speedup estimates in Table 4.7. The PARASITE results for the full window, unlimited processor case agree closely with some of the Type 3 estimates for the 4-iteration case, and are as much as 2 times smaller in some of the other cases. The discrepancies are greatest for the *scdac* and *digfi* circuits which are comparatively large, evenly partitioned circuits. These are the circuits that are most likely to have a large number of subcircuits with comparatively slowly changing signals in any given window. The variable time step integration algorithm will choose very long time steps and will consequently compute very few time points in these subcircuits, whereas the subcircuits with rapidly changing signals will require many time points. Consequently, the subcircuits with low signal activity cause a reduction in the available parallelism as compared with the Type 3 estimates. It should be noted that even though subcircuits with low

Table 4.7. PARASITE Speedup Estimates

Task Graph	Algorithm	Circuit	Processors 1	2	4	8	16	32	∞
T_{GS} or $\tilde{T}_{GS}$	FWT-GS	dvs	1.0	1.4	1.5	1.5	1.5	1.5	1.5
		dpla	1.0	1.7	2.1	2.1	2.1	2.1	2.1
		scdac	1.0	2.0	3.7	5.3	5.6	5.6	5.6
		ben2k	1.0	1.7	2.1	2.3	2.4	2.4	2.4
		digfi	1.0	2.0	3.3	3.8	3.9	3.9	3.9
	TPP-GS	dvs	1.0	1.8	2.3	2.4	2.4	2.4	2.4
		dpla	1.0	1.9	2.8	3.0	3.0	3.0	3.0
		scdac	1.0	2.0	3.9	7.0	9.4	9.8	9.8
		ben2k	1.0	1.9	2.8	3.1	3.2	3.2	3.2
		digfi	1.0	2.0	3.9	5.8	6.1	6.2	6.2
T_{GJ}	FWT-GJ	dvs	1.0	2.0	3.5	5.3	6.2	6.3	6.3
		dpla	1.0	2.0	3.8	6.6	8.1	8.2	8.2
		scdac	1.0	2.0	3.9	7.2	11.0	12.6	12.9
		ben2k	1.0	2.0	3.8	7.2	9.9	10.8	11.4
		digfi	1.0	2.0	3.9	7.6	13.7	19.5	22.2
	TPP-GJ	dvs	1.0	2.0	3.8	6.8	9.1	9.3	9.3
		dpla	1.0	2.0	3.9	7.1	10.6	12.1	12.1
		scdac	1.0	2.0	3.9	7.5	13.0	17.7	19.3
		ben2k	1.0	2.0	3.9	7.4	11.7	13.5	14.1
		digfi	1.0	2.0	4.0	7.8	14.5	24.2	33.2
$\tilde{T}_{GJ}$	FWT-GJ	dvs	1.0	1.9	3.1	4.3	5.0	5.2	5.2
		dpla	1.0	2.0	3.7	5.7	6.3	6.3	6.3
		scdac	1.0	2.0	3.8	6.4	8.0	8.4	8.5
		ben2k	1.0	2.0	3.9	7.0	9.2	10.0	10.4
		digfi	1.0	2.0	3.9	7.4	12.0	14.8	15.9
	TPP-GJ	dvs	1.0	2.0	3.8	6.7	8.6	8.8	8.8
		dpla	1.0	2.0	3.9	6.7	9.2	9.9	9.9
		scdac	1.0	2.0	3.9	7.4	11.1	13.2	13.8
		ben2k	1.0	2.0	3.9	7.4	11.5	13.1	13.6
		digfi	1.0	2.0	4.0	7.7	14.2	21.2	25.6

signal activity reduce parallelism, they do so only because the uniprocessor waveform relaxation algorithm already takes advantage of this situation by not computing unnecessary time points in these subcircuits.

The PARASITE results are tabulated as a function of the number of processors. For the Gauss-Seidel method with the full window technique, the speedups reach their maximum values very quickly as the number of processors are increased due to the severely limited parallelism. The Gauss-Jacobi method with the full window technique exhibits speedups close to the number of processors until about 8 processors, depending on the circuit.

The time point pipelining speedup estimates are necessarily greater than the corresponding full window speedups, since time point pipelining exposes greater parallelism, and overhead is neglected. The degree by which the time point pipelining speedups are greater than the full window technique speedups depends directly on the the window sizes. The windows are chosen by the automatic windowing algorithm of RELAX2.3, which tends to favor small windows in order to keep the number of iterations small. Larger windows would result in greater time point pipelining speedups compared to the full window technique using the same enlarged windows; but very large windows would cause the 1-processor reference time to be increased due to an increase in the number of iterations.

Since the uniprocessor run times are known in computing post-simulation estimates, it is possible to normalize Gauss-Jacobi speedups to Gauss-Seidel, as shown in Table 4.8. The speedups less than 1 for the single processor case reflect the normalization factor. Even though Gauss-Jacobi starts out with this speed disadvantage on 1 processor, the normalized Gauss-Jacobi estimates surpass the corresponding Gauss-Seidel estimates when the number of processors is sufficiently large, for both the full window technique and time point pipelining. Consequently, the Gauss-Jacobi method with time point pipelining offers the greatest potential speed of the four algorithms when the number of processors is large.

The break-even point between the Gauss-Seidel and Gauss-Jacobi methods is a function of the circuit being simulated. Table 4.9 shows which of the relaxation

Table 4.8. PARASITE Speedup Estimates Normalized to Gauss-Seidel

Task Graph	Algorithm	Circuit	Processors 1	2	4	8	16	32	∞
T_{GJ}	FWT-GJ	dvs	0.7	1.4	2.5	3.8	4.5	4.5	4.5
		dpla	0.8	1.6	3.0	5.2	6.4	6.5	6.5
		scdac	0.7	1.6	3.1	5.7	8.7	10.0	10.2
		ben2k	0.6	1.3	2.4	4.6	6.3	6.8	7.2
		digfi	0.7	1.4	2.7	5.3	9.6	13.6	15.5
	TPP-GJ	dvs	0.7	1.4	2.7	4.9	6.5	6.7	6.7
		dpla	0.8	1.6	3.1	5.6	8.4	9.6	9.6
		scdac	0.7	1.4	2.8	5.3	9.2	12.5	13.6
		ben2k	0.6	1.3	2.5	4.7	7.4	8.5	8.9
		digfi	0.7	1.4	2.8	5.4	10.1	16.9	23.2
$\tilde{T}_{GJ}$	FWT-GJ	dvs	0.7	1.4	2.2	3.1	3.6	3.7	3.7
		dpla	0.8	1.6	2.9	4.5	5.0	5.0	5.0
		scdac	0.7	1.4	2.7	4.5	5.7	5.9	6.0
		ben2k	0.6	1.3	2.5	4.4	5.8	6.3	6.6
		digfi	0.7	1.4	2.7	5.2	8.4	10.3	11.1
	TPP-GJ	dvs	0.7	1.4	2.7	4.8	6.2	6.3	6.3
		dpla	0.8	1.6	3.1	5.3	7.3	7.8	7.8
		scdac	0.7	1.4	2.8	5.2	7.8	9.3	9.8
		ben2k	0.6	1.3	2.5	4.7	7.3	8.3	8.6
		digfi	0.7	1.4	2.8	5.4	9.9	14.8	17.9

Table 4.9. Fastest Method Based on PARASITE Using Unaugmented Task Graphs and Time Point Pipelining

Circuit	Processors 1	2	4	8	16	32	∞
dvs	S	S	J	J	J	J	J
dpla	S	S	J	J	J	J	J
scdac	S	S	S	S	S	J	J
ben2k	S	S	S	J	J	J	J
digfi	S	S	S	S	J	J	J
Key: S=Gauss-Seidel, J=Gauss-Jacobi							

methods is faster for each circuit, using time point pipelining, as a function of the number of processors, based on the PARASITE estimates. Typically, larger circuits have a higher break-even point because these circuits have more subcircuits and greater Gauss-Seidel parallelism. But as demonstrated by the *scdac* circuit, circuit size alone is not a sufficient indicator of the break-even point. The Gauss-Jacobi method becomes preferable only when the Gauss-Seidel method does not produce enough concurrent work for the available supply of processors.

Table 4.10 summarizes the impact of the extra constraints of the augmented task graph on the Gauss-Jacobi speedups predicted by PARASITE. Since all the benchmark circuits are bidirectionally coupled, the loss in speedup resulting from the use of the augmented constraints in the full window case on unlimited processors is at most 50%, as proved in Chapter 2. The results in the table are in agreement with this theoretical result. If only a limited number of processors are available, then the effect of the augmented graph may be greatly diminished, depending on the size and structure of the circuit. In the full window case, the larger circuits *ben2k* and *digfi* show less than a 5% degradation in speedup on 8 processors, because there is sufficient work to keep the processors fairly busy even when the extra constraints are added. The smallest circuit *dvs* has very little parallelism, and the full impact of the extra constraints is evident on 8 processors in the full window case. The time point pipelining speedups are affected to a

Table 4.10. PARASITE Estimated Speedup Loss: $\tilde{T}_{GJ}$ vs. T_{GJ}

Method	Circuit	Processors	
		8	∞
FWT-GJ	dvs	18%	18%
	dpla	13%	23%
	scdac	21%	41%
	ben2k	4%	8%
	digfi	2%	28%
TPP-GJ	dvs	2%	6%
	dpla	5%	19%
	scdac	2%	28%
	ben2k	0%	3%
	digfi	0%	23%

lesser degree by the extra constraints. This is not surprising, due to the parallelism which the pipelining affords between adjacent tasks in the task graph. On 8 processors the decrease in speedup is no greater than 5% for even the small circuits, due to the availability of many concurrently executable tasks, even with the added constraints.

CHAPTER 5

FULL WINDOW TECHNIQUE IMPLEMENTATION

The choice of Gauss-Jacobi or Gauss-Seidel relaxation and the choice of the full window technique or time point pipelining both represent tradeoffs involving parallelism and parallel processing overhead. Parallel processing overhead is intimately related to the multiprocessor architecture, the parallel algorithm, and the details of the implementation of the algorithm on the multiprocessor. To study the performance of the different algorithms in an actual parallel processing environment, the algorithms were implemented in programs which run on an Alliant FX/8 multiprocessor. The FWT program, which is described in this chapter, is an implementation of the full window technique. The TPP program embodies the time point pipelining algorithm and is described in Chapter 7. Since TPP was derived from FWT, much of the discussion in this chapter applies also to TPP.

The Alliant FX/8 hardware and software environment is briefly described in the following section. Next, the locking mechanism for protecting the integrity of shared data, and the task system which controls the parallel execution of tasks are described; these features are common to both FWT and TPP. The implementation of parallel processing in FWT is then described. Finally, performance measurements are given and compared with the estimates produced by PARASITE.

5.1 The Multiprocessor

The Alliant FX/8 is an 8-processor mini-supercomputer [All86a]. Each processor is capable of executing scalar and vector instructions, with a peak rate of 5.9 Mflops when operating on 64-bit floating point operands. Each processor contains vector and scalar registers and an instruction cache. All the processors share a common main memory system which is accessed through a shared cache, as shown in Fig. 5.1. Although the Alliant machine is limited to 8 processors, the Cedar multiprocessor supercomputer, currently under development at the Center for Supercomputing Research and Development at the University of Illinois, will consist of multiple Alliants interconnected through a global shared memory [Kuc86]. Consequently, the results presented in this chapter for actual multiprocessor runs are for 8 or fewer processors, and the speedup estimates of previous chapters will be used for predicting the performance on future machines with more processors.

The Alliant computer runs a UNIX operating system and has a FORTRAN compiler [All85] which automatically vectorizes and parallelizes DO loops, based on its

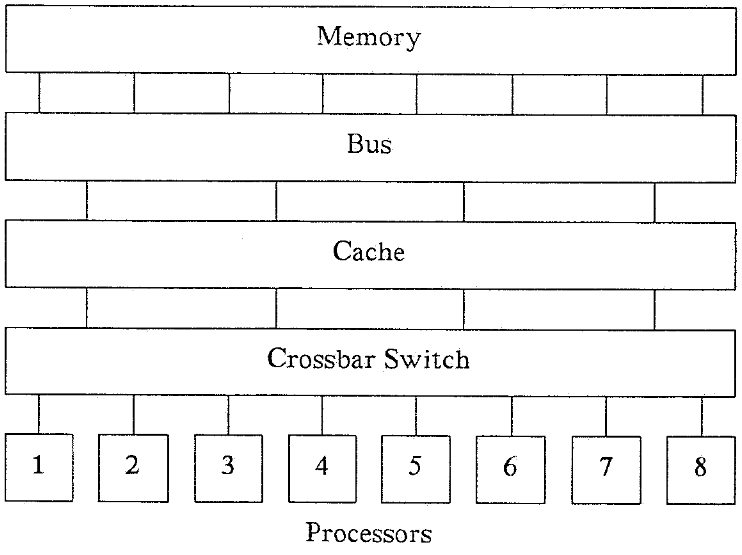


Figure 5.1. Alliant FX/8 architecture.

analysis of data dependencies. A compiler for the C language [All86b] is also available, but it does not perform automatic vectorization and parallelization. The FWT and TPP programs are derivatives of RELAX2.3, and all 3 of the programs are written in C. As a result, these programs are not automatically vectorized and parallelized by the compiler. However, even if the automatic parallelizing features of the FORTRAN compiler were available in the C compiler, the natural parallelism of waveform relaxation would not be recognized by the compiler.

Suppose that the C compiler included the automatic parallelization features, and that the main program of the uniprocessor waveform relaxation program is a realization of Algorithm 2.1, in which the subcircuit evaluation task is realized with a subroutine call. When the subroutine is called it is passed a pointer to the data structure for the subcircuit, and this structure contains pointers to other subcircuits which supply its input waveforms. When the main program is compiled, the compiler is not aware of how all these pointers will be used when the subroutine is executed, and the pointers are not even known at compile time because they are circuit dependent. Therefore, the compiler would decide to execute the subcircuit evaluation tasks serially, because the potential exists for arbitrary data interdependencies between the tasks. The interdependencies could result in improper operation if the tasks are allowed to execute concurrently.

A FORTRAN compiler directive is provided which allows the different iterations of a loop to be performed concurrently even if the loop contains a subroutine call. This could be used directly for the subcircuit loop in Algorithm 2.1 when Gauss-Jacobi is used. However, for Gauss-Seidel, the partial ordering of subcircuits within an iteration, as dictated by the task graph, must be enforced. These constraints are not readily recognized by the compiler because they arise from the interactions of executable statements and data structures which are built dynamically during program execution. Consequently, the programmer must assume responsibility for setting up a mechanism to control the parallel execution of waveform relaxation tasks.

The concurrent use of multiple processors is implemented in the C environment with the concurrent call feature. In a concurrent call, a C function is called a specified number of times, M. Each of the M invocations is assigned a unique index number and runs on a separate processor, provided $M \leqslant 8$. Since the function can test its index number and perform different actions based on its value, the processors may execute any independent instruction streams. When all of the concurrent function invocations terminate, control returns to the calling program which then continues running on a single processor.

Vector instructions are accessible from C programs through calls to library functions. However, vector instructions are not used in FWT and TPP, because most subcircuit are small in size and yield small vector lengths. In previous work, vector processing techniques have been used with the standard direct method algorithms applied to the entire circuit, in which case the potential for longer vector lengths exists. Even in this more favorable situation, the vector parallelism is limited due to the high degree of sparsity in circuit matrices, the irregular structure of the matrix, and the different equations which must be evaluated in different operating regions of the transistors.

The main memory of the Alliant FX/8 is shared by all the processors. One processor may communicate with another simply by writing to a memory location which is later read by the other processor. It is the programmer's responsibility to assure that one processor does not corrupt the data needed by another processor, and that one processor does not try to read data before the data are written by another processor. Data accesses by different processors to common memory locations can be synchronized using either global or local synchronization techniques.

The global synchronization technique is easiest to implement. A set of independent tasks are allowed to execute concurrently. Tasks which require input data from other tasks in the set are excluded from the set. When all the tasks are finished, a new set of independent tasks are started. Each set can consist of the tasks in one level of a task graph. However, performing global synchronizations between each level of the task graph does not exploit the full parallelism of the graph, as observed in the results of the Type 1 and Type 2 estimates of Chapter 4.

The local synchronization approach allows data to be exchanged between two processors without waiting for the other processors to reach a global synchronization point. Local synchronizations allow for the globally unsynchronized execution of a task graph as assumed in the Type 3 and PARASITE estimates of Chapter 4. Local synchronizations are accomplished on the Alliant using locks, which are based on the atomic test-and-set instruction.

5.2 Locks

Locks are typically used to protect shared data during critical portions of time during which the data may only be accessed by one processor. For example, in Algorithm 2.2 the global variable $unconv_k$ is accessed 3 times. Its value is read from memory, and after the value is decremented it is written back to memory. Then it is read again to test if its value is 0. In order to guarantee the proper operation of this algorithm, no other processor executing the algorithm concurrently may access $unconv_k$ during the time interval spanned by these 3 accesses. This rule can be enforced by associating a lock with the variable $unconv_k$, and by locking the lock before the first access and unlocking the lock after the last of the 3 accesses. If a processor tries to lock a lock which is already locked, it will be forced to wait until the lock is released by the other processor before it is allowed to acquire the lock and continue.

Locks are implemented with the atomic test-and-set machine instruction. This instruction is not directly accessible in a C language statement, but may be accessed through a function call or through embedded assembly language instructions. The FWT and TPP programs set locks using a LOCK macro, which is translated into either a function call or assembly language instructions prior to compilation. In either case, the LOCK macro causes the execution of the following algorithm, in which x is a lock variable.

Algorithm 5.1. LOCK(x)

```
repeat {
        test-and-set($x$)
        delay
} until (the test-and-set operation is successful)
```

The test-and-set operation is successful only if $x=0$, in which case it sets $x \leftarrow 1$. The testing of the value of x and the setting of its value to 1 are performed as an atomic operation to prevent two different processors from successfully setting the same lock simultaneously. The delay is added for performance reasons to be discussed below. Locks are unlocked in FWT and TPP by an UNLOCK macro. By convention, only the processor which locked a lock is allowed to unlock it. The following rather brief algorithm accomplishes the unlocking chore and is trivially implemented directly in C code.

Algorithm 5.2. UNLOCK(x)

```
$x \leftarrow 0$
```

When one processor tries to lock a lock which is already locked, it enters a tight loop in which it repeatedly tests the value of the lock until the lock is released by the other processor. This has two effects on performance. The most obvious and important effect is that the waiting processor does not do any useful work while it is waiting for the lock. A second effect is that the waiting processor generates cache traffic while repeatedly testing the lock, and thereby slows down the gainfully employed processors.

The cache competition caused by processors waiting on locks can be reduced by adding null operations in the loop to effect a delay between successive accesses of the lock. The null operations are presumably executed out of the local processor instruction cache without competing for access to the shared cache. As the delay in the loop is increased, the cache traffic is further reduced. However, increasing the delay also increases the average time that a waiting task will be delayed after a lock is released by another processor before realizing that the lock is available.

A locking operation which uses a function call is referred to as a *normal* lock, and the case of embedded assembly language is referred to as a *fast* lock. A fast lock which is successful on its first attempt executes only two machine instructions, the test-and-set and a conditional branch. A normal lock includes the overhead of the function call mechanism, which is significant compared to the time required for only two machine instructions. If the lock is not acquired successfully on the first attempt, the performance difference between fast and normal locks becomes less significant.

Normal locks are portable and robust with respect to revisions in the compiler. Fast locks do not share these advantages because the interface between the C code and assembly code is through a register which is assigned automatically by the compiler. When using many locking operations to control fine-grained accesses, the function call overhead can be significant, especially if the lock is acquired on the first try in most cases, since then only two machine instructions are executed. But, if the number of locking operations is a small percentage of the total number of operations, then the performance advantage of fast locks is insignificant. Due to the relatively large task granularity employed in FWT and TPP, the use of fast locks does not result in a significant performance improvement. Experiments with finer-grained tasks have demonstrated that fast locks can make a noticeable impact on performance in such cases.

A more fundamental observation concerning locks on the Alliant compared to some other machines is that even normal locks are quite fast. Therefore, when the level of concurrency can be increased by using more locks, each controlling a smaller group of shared variables, the resulting increase in the time spent performing locking operations will normally not be severe. Therefore, the general guideline applied to the use of locks in the FWT and TPP programs has been to keep the amount of code in locked sections small, even if this requires extra locking operations and extra lock variables.

5.3 Task System

A central fixture of the FWT and TPP programs is the task system which queues tasks, assigns queued tasks to processors, and terminates the parallel processing mode when all tasks are done. A central queue is used to hold tasks which are ready to execute. In some parallel processing architectures, some memory locations can be accessed more quickly from a particular processor or group of processors. In such systems, the assignment of tasks to processors may take the proximity of the required data into account. However, on the Alliant, all data in the cache and memory are equally accessible from any processor. Therefore, data proximity considerations are not appropriate in the task scheduler for the FWT and TPP programs on the Alliant. When a processor becomes available to begin working on a new task, the next appropriate task is taken from the queue and assigned to the processor.

Two complications are introduced by allowing one waveform relaxation iteration to begin before the preceding iteration is completed. One problem is that a task in a later iteration may become eligible for execution before it is determined that convergence occurs on a preceding iteration. Consequently, unnecessary tasks may be executed, and these tasks will use resources which would be better used for required tasks associated with earlier iterations. This problem is addressed by assigning each task a priority based on its iteration number. Tasks associated with lower iteration numbers are assigned higher priorities. Separate subqueues are maintained for each priority level. When the task system obtains a task from the queue to be executed, it selects a task from the nonempty subqueue with the highest priority.

The other problem arising from overlapping iterations is that convergence might be obtained while some tasks are in the middle of execution. For this reason, and also to accommodate some recoverable error conditions that can arise during the simulation such as waveform buffer overflow, a facility is provided to gracefully kill all tasks which are executing or queued and to prevent new tasks from being queued. Executing tasks periodically check a flag, and if it is set they terminate with their associated data structures in a state which is acceptable for continuing the simulation after the global synchronization.

The operation of the task system is summarized in the following algorithm outlines. The global variable *tq_count* is the number of tasks which are queued or are executing. The lock *queue_lock* protects the queue and *tq_count* . A task *tsk* of priority *p* is queued by the following algorithm:

Algorithm 5.3. Queue_Task (tsk, p)

```
if ($kill\_flag = FALSE$) {
        LOCK($queue\_lock$)
        append $tsk$ to subqueue $p$
        $tq\_count \leftarrow tq\_count + 1$
        UNLOCK($queue\_lock$)
}
```

The execution of tasks on all parallel processors is under the control of the parallel task controller, which is executed on each processor using the concurrent call mechanism.

Algorithm 5.4. Parallel_Task_Controller—runs on each processor

```
tsk ← NULL
while (tq_count > 0) {
        LOCK(queue_lock)
        if (there is a task on the queue) {
                tsk ← (task from lowest numbered subqueue)
                remove tsk from subqueue
        }
        UNLOCK(queue_lock)
        if (tsk ≠ NULL) {
                execute tsk
                tsk ← NULL
                LOCK(queue_lock)
                tq_count ← tq_count − 1
                UNLOCK(queue_lock)
        }
        else delay
}
```

Note that if the queue is empty, then the **if** clauses are bypassed and the operations in the loop consist of repeated accesses to a few global variables: *queue_lock*, *tp_count*, and the location in the queue data structure which indicates that the queue is empty. If several processors are without work and the queue is empty, this results in excessive cache traffic for these variables, which interferes with the operation of the working processors. The delay in the **else** clause is introduced to relieve the cache congestion in this case, in the same manner in which the delay was introduced in the LOCK routine. The value of the delay is optimized empirically.

The task killer is invoked by any task which determines that all other currently queued and executing tasks are unnecessary.

Algorithm 5.5. Kill_Tasks

```
LOCK(queue_lock)
kill_flag ← TRUE
remove all tasks from all subqueues
```

```
decrement *tq_count* by the number of tasks removed
UNLOCK(*queue_lock* )
wait until *tq_count* =1, (i.e., until all other tasks terminate)
```

5.4 RELmod

The starting point for the development of the FWT program was the RELAX2.3 program developed by Jacob White [Whi86]. RELAX2.3 is a uniprocessor waveform relaxation program. It was modified to produce another uniprocessor program known as RELmod, which contains additional research-oriented features. RELmod served as the basis for the FWT multiprocessor waveform relaxation program which uses the full window approach to parallelism. Finally, the FWT program was modified to create the TPP program which uses time point pipelining.

RELmod uses the partitioning, ordering, windowing, and numerical integration algorithms of RELAX2.3 without any substantive changes. The modifications incorporated into RELmod include corrections of bugs and features intended primarily for research use. Specifically, RELmod contains these features:

(a) The Gauss-Jacobi method, and hybrid Gauss-Jacobi/Gauss-Seidel methods are implemented.

(b) Several bugs are fixed, the most notable being a bug in the resetting of the error tolerances when starting a new window. This bug had a significant negative impact on the Gauss-Jacobi run times in some cases, where slow convergence led to repeated reductions in window size and repeated reductions in error tolerance which were never retracted.

(c) Overlapped partitioning of subcircuits is permitted as an option [Mok85].

(d) The subcircuit partitioning and ordering may optionally be specified manually, rather than being generated automatically.

(e) Window boundaries may optionally be specified manually, rather than being determined automatically.

(f) Optional output files may be requested containing the subcircuit partitioning and ordering, and the initial conditions used. These files are in a format acceptable for input to the program on a later run. This allows many transient analysis runs to be made without repeating the partitioning, ordering, and initial dc solution computations. Also, it facilitates experiments in which minor changes are made in the automatically generated partitioning.

(g) The rules for specifying periodic piecewise linear voltage sources are more user friendly.

Since RELmod uses the same numerical algorithms as FWT without any of the extra parallel processing code, it serves as an ideal reference to which the performance of FWT can be compared. Comparisons of the run times of RELmod and FWT indicate that the extra overhead of FWT on 1 processor increases the run time by less than 2%.

5.5 FWT

The FWT program uses RELmod as a base, and implements the full window technique for parallel waveform relaxation. It can be run on 1 to 8 processors of an Alliant FX/8. Parallelism is exploited only in the transient analysis phase of the program, since this is the most time consuming and the area of greatest potential payoff.

The basic idea behind the FWT program implementation is quite simple. A template of the augmented task graph $\tilde{T}$ is constructed for the specified relaxation method. In each window, the initial tasks are placed on the task system queue. The task system assigns tasks from the queue to available processors. As each task finishes execution, it checks its successor tasks and queues those for which all the input waveforms have been computed. Also, just before terminating, a task performs a convergence check on its own waveforms, and checks the accumulated convergence status of all other tasks in the iteration to see if convergence was obtained on that iteration. When convergence is detected, all executing and queued tasks are killed and a new window is started.

Within each window, the iterations are partitioned into iteration groups, such that each group contains κ consecutive iterations. A global synchronization is performed between each iteration group. The default value of κ is 6. There are several reasons for adding these global synchronization points. The primary reason is that it simplifies the implementation of the periodic reductions of the window size and error tolerance which occur when too many iterations are used. The second reason is that it allows for fixed limits to be placed on certain arrays which require a separate array element for each iteration of the group. Finally, it limits the number of unnecessary tasks, with iteration numbers greater than the converging iteration number, which may be executed before convergence is detected.

The added synchronization points between iteration groups can result in reduced parallelism. However, in most windows convergence is obtained in the first iteration group, and the extra synchronizations do not occur. Furthermore, the Type 3 speedup estimates in Tables 4.5 and 4.6 indicate that the parallelism is only a weak function of the number of iterations, especially when the number of iterations is greater than 4. Therefore, even if several groups of 6 iterations are required in a window, the speedup in each group will be about the same as could be obtained without the synchronizations between groups.

The algorithm for the transient analysis phase of FWT, using iteration groups, is outlined below. The current window boundaries are represented by t_a and t_b, k_{start} and k_{stop} are the first and last iterations of the current iteration group. The *global_converged* flag becomes *TRUE* when convergence is detected in all subcircuits on some iteration. For each subcircuit there are κ counters, one associated with each iteration of the group. Counter $unsat_{k,i}$ represents the number of unsatisfied precedence constraints for subcircuit evaluation task (k, i), *i.e.*, the number of predecessors in the task graph which have not finished execution.

Algorithm 5.6. FWT Transient Analysis

```
$t_a \leftarrow 0$
while ($t_a < t_f$) {                    /*window loop */
        choose $t_b$
        $k_{start} \leftarrow 1$
        $k_{stop} \leftarrow \kappa$
        $global\_converged = FALSE$
        repeat {                  /* iteration group loop */
                initialize $unsat_{k,i}$, for $k_{start} \leqslant k \leqslant k_{stop}$, $1 \leqslant i \leqslant n$
                queue initial tasks for iteration group
                Parallel_Task_Controller ()
                if ($global\_converged = FALSE$) {
                        $k_{start} \leftarrow k_{stop} + 1$
                        $k_{stop} \leftarrow k_{stop} + \kappa$
                        reduce integration error tolerance
                        reduce $t_b$
                }
        } until ($global\_converged = TRUE$)
        $t_a \leftarrow t_b$
        reinitialize integration error tolerance
}
```

The program runs on a single processor except when the parallel task controller is running, in which case N_{procs} processors are used, for a specified value of N_{procs} between 1 and 8.

The subcircuit evaluation tasks are executed under the control of the parallel task controller. The initial tasks are queued prior to turning control over to the parallel task controller. These are the tasks which have an in-degree of zero in the task graph, after all tasks of previous iteration groups are removed. Before terminating, a task updates the *unsat* counters of its successors and queues successor tasks which are ready to execute, as specified in the following algorithm.

Algorithm 5.7. Subcircuit Evaluation Task (k, i)

```
                /* Solve subcircuit */
Solve subcircuit $i$ on iteration $k$ over time interval $[t_a, t_b]$
                /* Check successors */
for each (successor $(k_{suc}, i_{suc})$ of $(k, i)$ in $\tilde{T}$) {
    LOCK($unsat\_lock_{i_{suc}}$)
    $unsat_{k_{suc}, i_{suc}} \leftarrow unsat_{k_{suc}, i_{suc}} - 1$
    if ($unsat_{k_{suc}, i_{suc}} = 0$) $queue\_task(k_{suc}, i_{suc})$
    UNLOCK($unsat\_lock_{i_{suc}}$)
}
                /* Check convergence */
if ($v_i^{(k)}$ matches $v_i^{(k-1)}$ within tolerance, for $t \in [t_a, t_b]$) {
    LOCK($unconv\_lock_k$)
    $unconv_k \leftarrow unconv_k - 1$
    if ($unconv_k = 0$) $conv \leftarrow TRUE$
    else $conv \leftarrow FALSE$
    UNLOCK($unconv\_lock_k$)
    if ($conv = TRUE$) {
        $global\_converged \leftarrow TRUE$
        $Kill\_Tasks()$
    }
}
```

5.6 Data Structures

The FWT program uses the augmented form of the task graph, $\tilde{T}$, which simplifies the management of data structures and reduces the required memory space. Since only one instance of any given subcircuit can be active at a time, most data structures associated with the subcircuit need only be allocated once, and may be reused by each task which is an instance of the subcircuit. The most important data structures which fall into this category include

(a) space for the matrix and vectors representing the linearized system of equations on each Newton iteration;

(b) the list of devices contained in the subcircuit, along with parameter values and pointers into the matrix which indicate where contributions from the model equations should be loaded into the matrix on each Newton iteration;

(c) relative pointers to successor tasks; and

(d) the time values and vectors of voltages, currents, and charges at the last few time points as required by the integration algorithm.

Some data structures, however, require separate copies corresponding to different iterations of a subcircuit, including

(a) the *unsat* counters, and

(b) waveform buffers which contain the time/voltage pairs for the time points computed at each node in the current window.

The number of *unsat* counters is κn, and these counters are simply allocated once and initialized at the beginning of each iteration group. The waveform buffers represent a larger investment in storage space.

Even though only one instance of each subcircuit will be active at any time, it is possible in general that all κ instances of the resulting waveforms of a given subcircuit may be required simultaneously by other subcircuits. Consider the example in Fig. 5.2 of a circuit which is not bidirectionally coupled. If the instances of subcircuits 1 and 2 require a small computation time compared to the instances of subcircuit 3, it is possible that all κ instances of subcircuits 1 and 2 will be finished before the first instance of subcircuit 3 finishes. Thus, while task (1, 3) is executing, the results of (1, 1) and (1, 2) are needed as inputs to (1, 3), and the results of all other instances of subcircuits 1

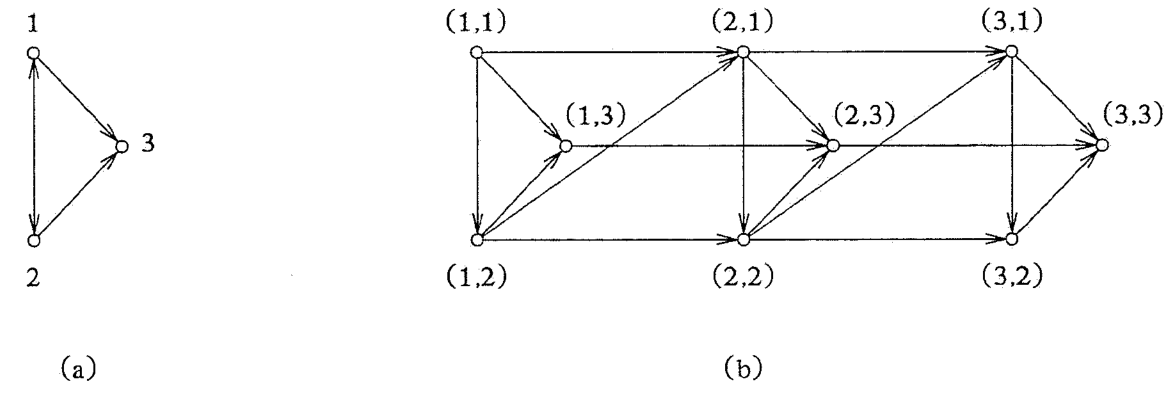


Figure 5.2. Waveform buffer example; (a) G , (b) $\tilde{T}_{GS,3}$.

and 2 must be saved for future use. To accommodate this situation, a provision must be made to have κ waveform buffers existing simultaneously for the nodes of subcircuits 1 and 2.

Sometimes, separate waveform buffers are not needed for each iteration. The next theorem provides a limit on the number of simultaneously required waveform buffers in the case of bidirectionally coupled circuits.

Theorem 5.1. *If G is bidirectionally coupled, then no more than 2 waveform buffers are needed for each node at any one time in the FWT program based on either $\tilde{T}_{GJ,\kappa}$ or $\tilde{T}_{GS,\kappa}$.*

Let (k,i) be any task in the task graph, and define a set Γ of tasks which are instances of subcircuit i such that

$$\Gamma = \{(m,i) : 1 \leqslant m \leqslant k-2\}. \tag{5.1}$$

When task (k,i) begins execution, it requires waveform buffers to store the results for its internal nodes. This is the first time that these waveform buffers are needed. If the contents of a waveform buffer are computed by task x , then the waveform buffer is no longer needed after all successors of x have terminated. Since (k,i) begins executing before any instance of i with a greater iteration number, it is sufficient to show that when (k,i) starts, all successors of tasks in Γ have already finished execution. If this is true, then every time a new waveform buffer is needed for a node, only one other waveform buffer for that node on a different iteration is required simultaneously—namely, the waveform buffer of the previous iteration. Equivalently, it is sufficient to show that there is a path to (k,i) from each successor of each task in Γ. Let (k_0,i_0) be any successor of any task in Γ. Note that $k_0 \leqslant k-1$. There is a path from (k_0,i_0) to (k,i) of the form

$$(k_0,i_0),(k_0,i),(k_0+1,i),\ \cdots,(k-1,i),(k,i)$$

or

$$(k_0,i_0),(k_0+1,i),\ \cdots,(k-1,i),(k,i).$$

All arcs in the path except the first are arcs which are added when the augmented graph graph is constructed from the unaugmented graph. The first arc must exist because (k_0,i_0) is a successor of an instance of i and the circuit is bidirectionally coupled. Hence, no more than 2 waveform buffers are needed for each node. Note that waveform buffers for iterations k and $k-1$ are required simultaneously in order to perform convergence checking.

To handle general circuits, the program must be able to provide separate waveform buffers for each node on each iteration, all existing simultaneously. This amounts to κN waveform buffers, where N is the total number of nodes in the circuit. However, for bidirectionally coupled circuits, only $2N$ waveform buffers are needed at any one time. In order to accommodate general circuits and not waste excessive memory space for bidirectionally coupled circuits, the FWT program dynamically allocates waveform buffers as needed. A pool of available waveform buffers is maintained on each processor. When a processor runs out of waveform buffers, it obtains an additional set of buffers by executing the dynamic memory allocation program. Although the dynamic memory allocator is executed in a critical section by at most one processor at a time, contention between different processors is minimized because most requests for waveform buffers are satisfied locally from the processor's own pool of waveform buffers. A counter is associated with each waveform buffer, indicating how many tasks, which have not yet terminated, require the use of the waveform buffer. The program is designed such that when the count reaches 0, the waveform buffer is freed by returning it to the pool of available buffers. This design has not been fully implemented in the current version of FWT. Currently, waveform buffers are not freed until the end of the iteration group. Due to the large virtual address space, this has not presented a problem in the simulation of the benchmark circuits.

5.7 Results

The performance results for the FWT program are given in Tables 5.1 and 5.2. The results are compared with the PARASITE results in Table 5.3. The FWT speedups on 8 processors are within 11% of the PARASITE estimates. The differences between the measured speedups and the estimates include the overhead factors and the estimate errors noted in Chapter 4. The FWT program sometimes generates slightly different window boundaries under multiprocessing conditions than in the uniprocessor reference run, which introduces an additional small error in the comparison. As a point of reference, the *scdac* runs using Gauss-Seidel used exactly the same window boundaries in all the multiprocessor and uniprocessor runs. The good agreement between the estimates and the actual results indicate that overhead plays a minor role in determining the performance of FWT on these benchmark circuits.

Table 5.1. FWT Gauss-Seidel Speedups

Circuit	Processors 1	2	4	8
dvs	1.0	1.4	1.4	1.4
dpla	1.0	1.6	2.0	2.0
scdac	1.0	2.0	3.6	4.9
ben2k	1.0	1.6	1.9	2.1
digfi	1.0	2.0	3.2	3.9

Table 5.2. FWT Gauss-Jacobi Speedups, Normalized to Gauss-Seidel

Circuit	Processors 1	2	4	8
dvs	0.7	1.3	2.2	3.1
dpla	0.8	1.6	2.9	4.0
scdac	0.7	1.4	2.6	4.5
ben2k	0.6	1.3	2.3	4.0
digfi	0.7	1.4	2.8	4.8

Table 5.3. Comparison of FWT and PARASITE on 8 Processors

Method	Circuit	Normalized Speedup FWT	PARASITE	Difference	Processor Utilization
Gauss-Seidel	dvs	1.4	1.5	7%	18%
	dpla	2.0	2.1	5%	25%
	scdac	4.9	5.3	8%	61%
	ben2k	2.1	2.3	9%	27%
	digfi	3.9	3.8	-3%	48%
Gauss-Jacobi	dvs	3.1	3.1	0%	55%
	dpla	4.0	4.5	11%	63%
	scdac	4.5	4.5	0%	80%
	ben2k	4.0	4.4	9%	79%
	digfi	4.8	5.2	8%	86%

The processor utilizations shown in Table 5.3 are the unnormalized FWT speedups divided by the number of processors. Consequently, these numbers do not include the time during which processors are utilized to perform those overhead computations which occur in the 8 processor case but not in the 1 processor case, and the time spent by processors waiting for access to shared resources.

5.8 Presimulation Selection of Gauss-Jacobi or Gauss-Seidel

When confronted with a circuit to simulate and a multiprocessor with a given number of processors on which to perform the simulation, the choice between the Gauss-Seidel and Gauss-Jacobi relaxation methods should take both the circuit structure and number of processors into account. For the benchmark circuits, the fastest of the two methods is indicated in Table 5.4, as a function of the number of processors, based on the performance results presented in Tables 5.1 and 5.2. In the typical situation for a given circuit, Gauss-Seidel is the faster method when the number of processors is small. As the number of processors is increased, a break-even point p_b is reached such that if more than p_b processors are used then the Gauss-Jacobi method is faster than Gauss-Seidel. The presimulation estimates of Chapter 4, together with rules of thumb obtained from performance data of other circuits, can be used to estimate p_b and therefore serve as a guide in selecting the relaxation method prior to performing the simulation.

At the break-even point, the Gauss-Seidel method typically has nearly reached its maximum possible speedup, whereas the normalized Gauss-Jacobi speedup is still increasing nearly linearly, with a slope close to the uniprocessor normalized speedup.

Table 5.4. Fastest Method vs. Number of Processors

Circuit	Processors			
	1	2	4	8
dvs	S	S	J	J
dpla	S	-	J	J
scdac	S	S	S	S
ben2k	S	S	J	J
digfi	S	S	S	J
Key: S=Gauss-Seidel, J=Gauss-Jacobi				

Despite the variety of types of circuits included in the set of benchmark circuits, the Gauss-Jacobi single processor normalized speedup is consistently close to 0.7. Therefore, at the break-even point, the normalized Gauss-Jacobi speedup is approximately $0.7p_b$. Since the Gauss-Seidel and Gauss-Jacobi speedups are equal at the break-even point, it follows that

$$p_b \approx \frac{S_{GS,est}}{0.7}, \tag{5.2}$$

where $S_{GS,est}$ is an estimate of the maximum Gauss-Seidel speedup on an unlimited number of processors. The maximum Gauss-Seidel speedup can be estimated by one of the presimulation estimation techniques of Chapter 4. If the Type 3 estimate is used, based on the assumption that 2 iterations will be used, then the break-even estimates of Table 5.5 result. These estimates agree quite well with the observed break-even points in Table 5.4.

It should be noted that the constant 0.7 used in the break-even estimate may in general depend on the partitioning and windowing algorithms. Any alterations to these algorithms would necessitate a reconsideration of the value of the constant. It is also possible that certain types of circuits will not agree with this choice of the constant. Experience with a larger number of circuits is required to determine if a single value for

Table 5.5. Presimulation Estimate of Break-even Point

Circuit	p_b
dvs	2.1
dpla	2.9
scdac	10.0
ben2k	3.3
digfi	7.1

the constant always produces a reasonable estimate, or if different classes of circuits each need a different value for the constant.

CHAPTER 6

TIME POINT PIPELINING IMPLEMENTATION

The implementation of time point pipelining in the TPP program is described in this chapter, and performance results are presented. The TPP program is based on the FWT program, with modifications to the procedures which determine when a task is eligible to execute and to the related data structures. The important implementation issues of time point pipelining are exposed by describing how they were addressed in the TPP program. The performance results show that TPP produces faster run times than FWT when there are sufficiently many processors to make use of the extra parallelism of time point pipelining. When too few processors are used, the extra overhead of TPP results in slower run times than FWT.

6.1 Algorithms

The coordination of parallel computations is significantly more complex in TPP than FWT. In FWT, each task consists of the evaluation of a subcircuit over an entire window on a single iteration, and the task graph for an iteration group is known a priori. The eligibility of each task for execution is monitored simply through its *unsat* counter, which contains the number of incoming arcs in the task graph from tasks which have not yet finished executing. In TPP, computations are coordinated at the subtask level, where a subtask consists of the evaluation of a subcircuit at a single time point on a single iteration. The subtasks cannot be identified prior to beginning the iteration group, because the variable time steps used by the integration algorithm depend on the waveforms which are computed during the iteration group. Consequently, the subtask graph cannot be constructed a priori, and the use of *unsat* counters to monitor the eligibility of individual subtasks for execution is not feasible. Instead, the subtasks to be executed and the precedence constraints between subtasks must be determined dynamically during the solution process, based on the known full window task graph and based on the actual time point values selected by the integration algorithm.

In addition to the time point subtasks, TPP defines an initialization subtask which precedes the first time point computation in each subcircuit evaluation task, and a convergence checking subtask which follows the last time point computation in each task. The coordination of the execution of all the subtasks is facilitated by a set of control variables. Each of the κn subcircuit evaluation tasks of an iteration group is allocated a set of control variables, {*status* , *t_done* , *t_next* , *t_ready* , *waiting_for* , *lock* }. The control variables are initialized prior to starting each iteration group. The meanings of the variables are summarized below.

status :

The *status* variable has the value *UNINITIALIZED* if the initialization subtask has not been executed yet. Its value is *INITIALIZED* if the initialization is completed and the last time point of the window has not yet been computed. After the last time point is computed its value is set to *INTEGR_DONE* .

t_done :

The *t_done* variable represents the time through which the subcircuit has been solved in the iteration, and it is initialized to t_a , the initial time in the window.

t_next :

In most cases, *t_next* represents the time of the next time point to be computed by the task, that is *t_done* =*t_next* +h where h is the step size determined from the local truncation error of the integration method. More generally, *t_next* is the time such that if all predecessor tasks have progressed at least to *t_next*, then the task is eligible to execute its next subtask. The initial value of *t_next* is $t_a + \delta$, where δ is smaller than a minimum time step. Consequently, as soon as all predecessors of a task compute at least one time point, the task becomes eligible to execute its initialization subtask.

t_ready:

The value of *t_ready* is a time through which all input waveforms of the subcircuit evaluation task are guaranteed to be valid. The *t_ready* variable is always updated in synchronization with *waiting_for*.

waiting_for:

If a task is not queued or running, then *waiting_for* points to one of its predecessors which must advance before the task becomes eligible for execution. Otherwise, if a task is queued or running, then *waiting_for* is *NULL*, indicating that it is not waiting for data from another task. When a task is not queued or running, the predecessor indicated by *waiting_for* is responsible for updating the task's *t_ready* and *waiting_for* variables, and for queuing the task when it is ready to execute. Note that the *waiting_for* predecessor can transfer this responsibility to another predecessor by modifying *waiting_for*.

lock:

This is a lock which is used to synchronize updates of the control variables, when explicit synchronization is necessary.

Much of the infrastructure of FWT is used without change in TPP, including all the task system algorithms presented in Chapter 5. The basic transient analysis algorithm outlined in Algorithm 5.6 is also applicable to TPP, with the addition of the control variable initializations. The FWT subcircuit evaluation task of Algorithm 5.7, which embodies all the parallel computations, is modified for TPP. Although the numerical computations which it performs are essentially unchanged, the control of the computations is different. The subcircuit evaluation task is partitioned into subtasks which are performed sequentially. After each subtask is executed, its successor tasks are checked. Each successor which is ready to execute at least one subtask is queued. If a task runs out of input data before all the subtasks are executed, then the task terminates prematurely. It is requeued later by its *waiting_for* predecessor when sufficient input waveforms become available to allow the computation of the next subtask. An outline of the subcircuit evaluation task in TPP is given below.

Algorithm 6.1. TPP Subcircuit Evaluation Task (k, i)

```
                    /* initialization*/
    $terminate \leftarrow FALSE$ ; $cont\_integration \leftarrow FALSE$
    if ($status_{k,i} = UNINITIALIZED$) {
        initialize data structures
        pick next time value, $t\_next_{k,i}$
        $status_{k,i} \leftarrow INITIALIZED$
        LOCK($lock_{k,i}$)
        update $t\_ready_{k,i}$ and $waiting\_for_{k,i}$
        if ($waiting\_for_{k,i} \neq NULL$) $terminate \leftarrow TRUE$
        UNLOCK($lock_{k,i}$)
    }
                    /* integration */
    if (($status_{k,i} = INITIALIZED$) and ($terminate = FALSE$)) $cont\_integration \leftarrow TRUE$
    while ($cont\_integration = TRUE$) {
        compute time point at time $t \in (t\_done_{k,i}, t\_next_{k,i}]$
        $t\_done_{k,i} \leftarrow t$
        pick next time value, $t\_next_{k,i}$
        $Check\_Successors(k, i)$
        if ($t\_done_{k,i} \geqslant t_b$) {
            $status_{k,i} \leftarrow INTEGR\_DONE$
            $cont\_integration \leftarrow FALSE$
        }
        else if ($t\_next_{k,i} > t\_ready_{k,i}$) {
            LOCK($lock_{k,i}$)
            limit $t\_next_{k,i}$
            update $t\_ready_{k,i}$ and $waiting\_for_{k,i}$
```

```
                if ($waiting\_for_{k,i} \neq NULL$) $cont\_integration \leftarrow FALSE$
                UNLOCK($lock_{k,i}$)
            }
    }
                /* convergence checking */
    if ($status_{k,i} = INTEGR\_DONE$) check convergence
```

A successor check is performed after each time point is computed. In addition to the fact that successor checks are performed many more times in TPP than in FWT, the successor check algorithm is more complicated in TPP, because the subtask graph is not known a priori. The successor checking algorithm is summarized as follows:

Algorithm 6.2. Check_Successors (k, i)

```
    if (less than 4 time points have been computed in $(k,i)$) $t\_ref \leftarrow -1$
    else $t\_ref \leftarrow$ (the time of the third previous time point in $(k,i)$)
    for each (successor $(k_s, i_s)$ of $(k,i)$) {
        $queueit \leftarrow FALSE$
        LOCK($lock_{k_s,i_s}$)
                /* 3-step check */
        if (($waiting\_for_{k_s,i_s} \neq NULL$) and ($t\_ref > t\_done_{k_s,i_s}$) and
                ($t\_next_{k_s,i_s} > t\_done_{k,i}$)) {
            $t\_next_{k_s,i_s} \leftarrow t\_done_{k,i}$
            $waiting\_for_{k_s,i_s} \leftarrow (k,i)$
        }
                /* queuing check */
        if (($waiting\_for_{k_s,i_s} = (k,i)$) and ($t\_next_{k_s,i_s} \leq t\_done_{k,i}$)) {
            update $t\_ready_{k_s,i_s}$ and $waiting\_for_{k_s,i_s}$
            if ($waiting\_for_{k_s,i_s} = NULL$) $queueit \leftarrow TRUE$
        }
        UNLOCK($lock_{k_s,i_s}$)
        if ($queueit = TRUE$) queue task $(k_s, i_s)$
    }
```

Both Algorithms 6.1 and 6.2 perform updates of the t_ready and $waiting\_for$ control variables, using the following algorithm:

Algorithm 6.3. **Update** *$waiting\_for_{k,i}$* **and** *$t\_ready_{k,i}$*

```
    $(k_p, i_p) \leftarrow$ (a predecessor of $(k,i)$ with minimum $t\_done_{k_p,i_p}$)
    $t\_ready_{k,i} \leftarrow t\_done_{k_p,i_p}$
    if $(t\_ready_{k,i} \geqslant t\_next_{k,i})$ $waiting\_for_{k,i} \leftarrow NULL$
    else $waiting\_for \leftarrow (k_p, i_p)$
```

6.1.1 Determining and modifying *t_next*

Except during initialization and convergence checking, $t\_next_{k,i}$ is the tentative time value of the next time point to be computed by task (k,i). The actual time value of the next time point may turn out to be less than $t\_next_{k,i}$, if, after the initial attempt to compute the time point, the step size is reduced due to excessive local truncation error or excessive Newton iterations. The final value of $t\_next_{k,i}$ just prior to the first attempt to compute the time point is given by

$$t\_next_{k,i} = \min\{t\_done_{k,i} + h,\ t_b,\ t_{limit}\}. \tag{6.1}$$

In the first expression, h is the time step selected by the integration algorithm in its attempt to produce a local truncation error which will be approximately equal to, but not greater than, the specified error tolerance. The t_b bound is simply the upper window boundary. The t_{limit} bound has the effect of limiting the step size in (k,i) to be no greater than 3 steps of any of its predecessors. The reason for this limit is that the local truncation error estimate is based on derivatives of the waveforms computed from divided differences of previous time points. A sudden change in a circuit input waveform, or a circuit nonlinearity, may cause a sudden change in a subcircuit input waveform which cannot be anticipated by looking only at the past history of the waveform. By forcing the subcircuit to be evaluated at least once for every 3 input steps, these sudden unanticipated transitions will not be skipped over by inappropriately long steps based only on estimated truncation errors.

In uniprocessor waveform relaxation and in the FWT program, (6.1) can be evaluated as soon as (k,i) computes the time point at $t\_done_{k,i}$, because all predecessor waveforms are guaranteed to be available for the entire window. However, in TPP, the input waveforms may not be available beyond $t\_done_{k,i}$. Therefore, following the computation of the time point at $t\_done_{k,i}$, an initial value is computed for $t\_next_{k,i}$ based on the information available at that time. As more input waveform points become available in the future, it is necessary to reduce the value of $t\_next_{k,i}$ if one of the inputs takes 3 steps in the interval $(t\_done_{k,i}, t\_next_{k,i})$.

One possible approach for handling this situation in TPP would be to leave $t\_next_{k,i}$ unchanged until all the predecessor waveforms become available through $t\_next_{k,i}$, at which time task (k,i) would become eligible to compute its next time point. Then all the information would be available to reduce $t\_next_{k,i}$ if necessary before actually computing the time point. This approach is not used in TPP, because it can have a significant negative impact on the time point pipelining parallelism as demonstrated in the following example.

Suppose all the subcircuits start in a dc steady state at the beginning of a window. The truncation error estimate will indicate that an arbitrarily long time step can be taken, and the t_next values for the first time point of all tasks will be t_b, based on the available information at initialization time, except for those subcircuits connected to external voltage sources which make transitions in the window. The subcircuits connected to the voltage sources may compute many time points before reaching the end of the window. However, if the t_next values of their successors are not modified, the successors will be prevented from computing concurrent time points during this interval because they will be waiting for their predecessors to reach $t\_next_{k,i} = t_b$.

This problem is avoided in TPP by checking the *t_next* values of all successors after each time point is computed. The appropriate updating of successors is done in the "3-step check" section of Algorithm 6.2, for those successors which are not running or queued at the time of the successor check. Tasks which are running or queued are responsible for maintaining their own *t_next* values.

6.1.2 Synchronization of control variables

Some of the control variables may be accessed and updated by different tasks, and therefore special precautions are required to assure their integrity. Parallel accesses to the control variables of a task are synchronized through the use of the *lock* and *waiting_for* control variables. When *waiting_for* $=$*NULL*, the owner task is responsible for maintaining its own control variables, and no other task may modify them. When *waiting_for* $\neq$*NULL*, then only tasks other than the owner may modify the control variables, but they must do so in a critical section protected by the *lock* control variable.

When the owner task modifies its own control variables, it is normally not necessary to acquire the *lock* first, because *waiting_for* $=$*NULL* and no other task is eligible to modify the variables. However, when the owner task changes its *waiting_for* value to non-*NULL*, the *lock* must be used, and all the status variables must be up to date at the moment that the *lock* is released. When the *lock* is released, the responsibility for updating the task's control variables is transferred to the task's predecessors. In particular, *t_ready* must be up to date in order for the *waiting_for* predecessor to be able to properly queue the task when sufficient time points are available. And *t_next* must be up to date based on all available input data points, because only those data points computed by predecessors in the future will affect *t_next* through the successor check. The use of *lock* and *waiting_for* to synchronize accesses to the control variables is shown in

Algorithms 6.1 and 6.2.

6.1.3 Convergence checking

Although not shown explicitly in Algorithm 6.1, complications arise in scheduling the convergence checking subtask in TPP because TPP uses the unaugmented task graphs, as opposed to the augmented task graphs used by FWT. The reason for using the unaugmented task graphs is discussed in the next section on data structures. In this section, the impact of the unaugmented task graph on the convergence checker is considered.

The convergence checking subtask referenced in Algorithm 6.1 is identical in content to the convergence checking portion of Algorithm 5.7. However, extra precedence constraint checks must be performed in the TPP program. Recall that the unaugmented task graph does not include the dependency of (k,i) on $(k+1,i)$. Therefore, it is possible that task $(k+1,i)$ will finish computing its last time point before task (k,i) computes its last time point. This situation can arise in Gauss-Jacobi relaxation even for bidirectionally coupled circuits. When this situation occurs, task $(k+1,i)$ may not proceed immediately to perform a convergence check after computing its last time point. Instead, it must suspend execution, to be requeued later by task (k,i).

6.2 Data Structures

In the FWT program, a subcircuit may be active in only one iteration at a time, due to the use of the augmented task graph $\tilde{T}$. In time point pipelining, a subcircuit may be active in more than one iteration at a time even if the augmented task graph is used. In fact, Gauss-Jacobi time point pipelining requires that a subcircuit be allowed to be active in different iterations simultaneously, because this is the only source of additional parallelism which is exposed by time point pipelining compared to the full window technique. Gauss-Seidel time point pipelining also benefits from allowing a subcircuit to be active in different iterations simultaneously, even if the circuit is bidirectionally coupled.

The reason for using the augmented task graph in FWT was to avoid the necessity of duplicating data structures for different iterations of a subcircuit. In TPP, the data structures must be duplicated regardless of which form of the task graph is used. Therefore, TPP uses the unaugmented form of the task graph, which offers the potential of greater parallelism than the augmented form. The PARASITE estimates of Table 4.8 indicate that the unaugmented graph will not result in significantly better performance than the augmented graph when only 8 processors are used for the benchmark circuits. If more processors are used, the benefits of the unaugmented graph are more significant.

In order to allow different iterations of a subcircuit to be active simultaneously, the TPP program allocates separate copies of those data structures which are used and modified during the simulation of a subcircuit. This results in an increase in the amount of required memory space by a factor of κ for this class of data structures compared to the FWT program. The contents of some data structures are not changed during the simulation of subcircuits and these structures need not be duplicated. Consequently, the overall increase in required memory space is less than a factor of κ.

Those data structures for which each subcircuit evaluation task is allocated a separate copy include

(a) space for the matrix and vectors representing the linearized system of equations on each Newton iteration;

(b) pointers into the matrix which indicate where terms computed from the model equations of each circuit element should be added to a matrix element;

(c) pointers to successor and predecessor tasks;

(d) the time values and the node voltages, currents, and charges at the last few time points as required by the integration algorithm;

(e) the TPP control variables; and

(f) pointers to the waveform buffers for all internal nodes and source waveforms which affect the subcircuit on the specific iteration.

The subcircuit element values, model parameter values, and other invariant data describing a subcircuit are maintained in a single copy for each subcircuit.

The approach of preallocating separate data structures for each task is memory intensive and algorithmically simple compared to some other schemes which could be used. Approaches which are more conservative of memory may be preferable for cases where the virtual address space is not large enough, or where the real memory size is not large enough and the resulting page misses encountered by the virtual memory system cause a degradation in performance. One of the more conservative approaches to memory usage for the subcircuit matrices will be briefly outlined, although the approach is not implemented in TPP.

If p processors are used, and if $p \ll \kappa n$, then a significant reduction in memory usage can be achieved by allocating space for only one matrix per processor. The space for each matrix must be large enough to accommodate the largest matrix of any subcircuit. When a subtask executes on a processor, it uses the matrix space associated with that processor. This is feasible since no data stored in a matrix by a subtask are needed by any other subtask; when a subtask finishes, the contents of its matrix may be discarded. The problem with this approach concerns the handling of pointers to matrix locations. When circuit elements are evaluated, their contributions are loaded into the matrix through precomputed pointers. If the destination matrix is not known in advance for a given task, then either p sets of precomputed matrix pointers are needed, one for each possible destination matrix, or the pointers can be computed dynamically when the terms are added to the matrix. If p is large, then the use of p sets of matrix pointers results is too great of an additional memory requirement compared to the memory saved by eliminating the matrices. The dynamic computation of matrix pointers can be accomplished by adding a precomputed offset to the address of the matrix origin when each term is added to the matrix. In this case, each matrix load operation requires one extra address addition.

Matrix pointers are also used to indicate the structure of matrices stored in sparse form. In FWT and TPP, these pointers are computed once in the presimulation phase of the program. The pointers are used repeatedly during the solutions of the sparse systems of equations. In the scheme where only one matrix is allocated per processor, the pointers representing the matrix structure could be determined once and stored in a master copy of the matrix for each subcircuit. Then when a task needs to use a matrix, it could copy its matrix structure pointers while adding an offset to account for the location of the specific matrix space to be used in memory. This results in the added requirement of one copy of the matrix structure for each subcircuit, plus a computational cost in initializing the matrix each time it is assigned to a particular processor. In the TPP program, these extra computations are avoided by using the more memory intensive approach of preallocating one matrix for each subcircuit evaluation task. The matrix pointers for each copy of the matrix are computed once in the presimulation phase of the program, and they are used repeatedly in each iteration group.

6.3 Results

The performance of the TPP program was measured for the benchmark circuits, and the results are presented in Table 6.1 for the Gauss-Seidel method and Table 6.2 for the Gauss-Jacobi method. The results on 8 processors are compared with the PARASITE estimates in Table 6.3. The TPP results are within 21% of the ideal PARASITE estimates, reflecting the higher degree of overhead compared to the FWT results which are within 11% of the ideal speedups. The comments in Chapter 5 concerning the factors which influence the comparison with the PARASITE estimates are

Table 6.1. TPP Gauss-Seidel Speedups

Ckt	Processors			
	1	2	4	8
dvs	1.0	1.7	2.3	2.3
dpla	0.9	1.7	2.5	2.9
scdac	0.9	1.8	3.4	5.9
ben2k	0.9	1.7	2.3	2.7
digfi	0.9	1.7	3.4	4.6

Table 6.2. TPP Gauss-Jacobi Speedups, Normalized to Gauss-Seidel

Ckt	Processors			
	1	2	4	8
dvs	0.7	1.3	2.4	4.0
dpla	0.7	1.4	2.8	4.7
scdac	0.7	1.3	2.5	4.5
ben2k	0.6	1.1	2.1	3.7
digfi	0.6	1.3	2.5	4.5

Table 6.3. Comparison of TPP and PARASITE on 8 Processors

Method	Circuit	Normalized Speedup			Processor Utilization
		TPP	PARASITE	Difference	
Gauss-Seidel	dvs	2.3	2.4	4%	30%
	dpla	2.9	3.0	3%	39%
	scdac	5.9	7.0	16%	74%
	ben2k	2.7	3.1	13%	37%
	digfi	4.6	5.8	21%	62%
Gauss-Jacobi	dvs	4.0	4.9	18%	71%
	dpla	4.7	5.6	16%	79%
	scdac	4.5	5.3	15%	87%
	ben2k	3.7	4.7	21%	80%
	digfi	4.5	5.4	17%	87%

applicable here as well.

6.4 Presimulation Selection of TPP or FWT

The fastest method as a function of the number of processors is given in Table 6.4, based on the results in Tables 5.1, 5.2, 6.1, and 6.2. In Chapter 5 it was shown that the presimulation speedup estimates can be used to select between the Gauss-Seidel and Gauss-Jacobi methods when the full window technique is used. Time point pipelining has at least as much available parallelism as the full window technique, and any presimulation selection technique that does not take overhead factors into consideration will always favor time point pipelining over the full window technique. However, it is apparent from Table 6.4 that the full window technique is sometimes faster than time point pipelining because of its lower overhead. For best performance, time point pipelining should only be used in those cases where the full window technique does not have sufficient parallelism to keep the processors very busy. This suggests that estimates of processor utilization for the full window technique may be useful in selecting between the full window technique and time point pipelining.

Table 6.4. Fastest Method vs. Number of Processors

Circuit	Processors			
	1	2	4	8
dvs	FS	TS	TJ	TJ
dpla	FS	TS	FJ≈TJ	TJ
scdac	FS	FS	FS	TS
ben2k	FS	TS	TS=FJ	FJ
digfi	FS	FS	TS	FJ
Key: S=Gauss-Seidel, J=Gauss-Jacobi, F=FWT, T=TPP				

A procedure is now presented for selecting the best of the four methods prior to the simulation of a given circuit on a given number of processors. First, the choice between the Gauss-Seidel and Gauss-Jacobi relaxation methods is made based on the estimate of p_b as described in Chapter 5. Next, the potential processor utilization for the full window technique is computed by dividing the speedup estimate by the number of processors. If the potential utilization is larger than some threshold u_b, then the full window technique is used; otherwise time point pipelining is used. The procedure is specified in detail in the following algorithm. The circuit is assumed to be given, p is the number of processors to be used, and u_b is a predetermined constant.

Algorithm 6.4. Presimulation Selection

```
if ($p = 1$) {
        $r\_method \leftarrow GS$
        $p\_method \leftarrow FWT$
}
else {
        Compute $S_{GS,est}$, the Type 3 speedup estimate for Gauss-Seidel using
                the full window technique, assuming a 2-iteration augmented
                task graph and unlimited processors.
        $p_{b,est} \leftarrow S_{GS,est} / 0.7$
        if ($p < p_{b,est}$) $r\_method \leftarrow GS$
        else $r\_method \leftarrow GJ$
        Compute $S_{r\_method,est}$, the Type 3 speedup estimate for the full window
                technique using the relaxation method specified by $r\_method$,
                assuming a 2-iteration augmented task graph and unlimited
                processors.
        $u \leftarrow S_{r\_method,est} / p$
        if ($u \geqslant u_b$) $p\_method \leftarrow FWT$
        else $p\_method \leftarrow TPP$
}
```

At the conclusion of the algorithm, the relaxation method is given by *r_method*, and the choice between the full window technique and time point pipelining is given by *p_method*.

The potential utilization represented by u is the fraction of time that the processors would have to be busy to achieve speedup $S_{r\_method,est}$ on p processors, where $S_{r\_method,est}$ is the speedup of the full window technique assuming an unlimited supply of processors. If $u < 1$, then the processors will only be active part of the time. In this case, the use of time point pipelining is appropriate to increase the processor utilization and the speedup. If u is slightly greater than 1, then the processors will be busy all the time if the computational load is perfectly balanced between the different processors. However, the nonuniformities in task sizes and precedence constraint patterns result in imperfect load balancing. Consequently, there will still be certain times when the processors run out of available tasks. Therefore, time point pipelining should be beneficial in this case also. If $u >> 1$ then the full window technique generates much more concurrent work than the processors can handle, and consequently time point pipelining should not be used because it will just add more overhead. The break-even value of u, where the full window technique and time point pipelining have about the same performance is given by u_b. The value of u_b is expected to be greater than, but not much greater than 1.

In order to determine if there exists a value of u_b which would result in correct selections of the fastest algorithms, the values of u have been tabulated in Table 6.5, for the indicated relaxation methods. The value $u_b = 1.7$ is found to produce good results. With this choice of u_b, the presimulation selection algorithm chooses the methods shown in Table 6.6. In all cases except for one, the fastest method (or one with nearly the same performance) is chosen. In the one exceptional case, the chosen method is 8% slower than the fastest method.

These results demonstrate that a simple analysis of the task graphs can be used to get some indication of the relative performance of different methods prior to

Table 6.5. Presimulation FWT Potential Utilization Estimates

Circuit	Processors			
	1	2	4	8
dvs	1.5 (S)	0.8 (S)	1.6 (J)	0.8 (J)
dpla	2.0 (S)	1.0 (S)	1.6 (J)	0.8 (J)
scdac	7.0 (S)	3.5 (S)	1.8 (S)	0.9 (S)
ben2k	2.3 (S)	1.2 (S)	1.8 (J)	0.9 (J)
digfi	5.0 (S)	2.5 (S)	1.3 (S)	4.7 (J)
Key: S=Gauss-Seidel, J=Gauss-Jacobi				

Table 6.6. Presimulation Prediction of Fastest Method

Circuit	Processors			
	1	2	4	8
dvs	FS	TS	TJ	TJ
dpla	FS	TS	TJ	TJ
scdac	FS	FS	FS	TS
ben2k	FS	TS	FJ	TJ*
digfi	FS	FS	TS	FJ
Key: S=Gauss-Seidel, J=Gauss-Jacobi, *=Disagrees with Table 6.4				

simulation. Details of the presimulation selection algorithm may have to be modified if significant changes are made in the implementation details of the simulation algorithms or in the types of circuits being simulated. In particular, the Gauss-Jacobi/Gauss-Seidel ratio of 0.7 used in the p_b estimate, and the u_b value of 1.7 are empirically determined constants obtained based on the implementations in the FWT and TPP programs as applied to the benchmark circuits. For the benchmark circuits, which represent a variety of types of MOS digital circuits, the presimulation selection method produces good results using the previously mentioned constants.

CHAPTER 7

LATENCY

The number of computations required to solve a set of circuit equations over a time interval can be reduced by exploiting latency. Latency exploitation has been used in a variety of forms in several circuit simulators [Nag75, Yan80, Whi86, Sal87a, Sak81, Cox87, Rab76]. Latency is exploited by recognizing situations in which a new solution point will match some previous solution point. If this determination can be done cheaply in advance of actually solving for the new solution point, then the computations to solve for the new solution point can be avoided. Two types of latency are addressed in RELAX2.3: time latency and iteration latency. This chapter considers the use of these types of latency in the parallel waveform relaxation algorithms.

7.1 Time Latency

A subcircuit is said to be *time latent* in time interval $[t_1, t_2]$ if the inputs to the subcircuit are constant in the interval and the subcircuit is in a dc steady state at time t_1. In this situation it is not necessary to compute the voltages for any time point in the interval $(t_1, t_2]$ since the values will be identical to those at t_1. The subcircuit is determined to be in a dc steady state at t_1 by observing that the subcircuit's internal voltages and charges are constant and the currents flowing into capacitors are zero, within some tolerance. Once this determination is made, the end of the time latency interval can be found by searching for the time at which the first change occurs in an input waveform of the subcircuit.

Even without explicitly handling time latency, the basic waveform relaxation algorithm automatically exploits time latency and slowly changing signals by choosing the time steps independently for each subcircuit. Therefore, the advantage of explicitly managing time latency is not as great for waveform relaxation compared to other solution techniques which do not use multirate integration. The time latency algorithm may be more efficient in placing the next time point exactly at the point in time where the first input change occurs, whereas the time step computed based on the truncation error does not look ahead to future input changes, except indirectly through the 3-step check (see Algorithm 6.2).

A danger of time latency exploitation is that false latency detection can occur if the tolerances are too large. This is especially important in linear circuits, where decaying exponential waveforms satisfy the latency criteria before actually reaching their final values. In digital circuits, the likelihood of false latency detection is reduced because of the sharp nonlinearities which clamp signals to steady state values relatively quickly after transitions.

7.2 Iteration Latency

Subcircuit i is said to be *iteration latent* on iteration k if all the input waveforms of subcircuit evaluation task (k, i) are identical to the inputs of task $(k-1, i)$, within some tolerance. In this situation it is not necessary to solve the subcircuit on iteration k since the solution will be almost identical to the solution of iteration $k-1$. In previous work, iteration latency has been called *partial waveform convergence*. This is because iteration latency most commonly occurs when some of the subcircuits converge to the solution while the rest of the circuit requires additional iterations. After the first subcircuits converge, they become iteration latent for the remaining iterations while the other subcircuits continue to be evaluated.

In the Gauss-Jacobi method, a signal change originating in one subcircuit may require some number of iterations j to reach subcircuit i, because signals propagate through the subcircuit graph at the rate of one subcircuit per iteration. If subcircuit i is in a dc steady state at the beginning of the window, it will be iteration latent for iterations 1 through $j-1$. In this scenario, the terminology *partial waveform convergence* is inappropriate, because the latency is not due to convergence. The terminology *iteration latency* is more basic, and covers both of the situations described above.

Iteration latency can occur over an entire window, or over the first part of a window. Let I_{k-1} and I_k be the sets of input waveforms to the subcircuit when the subcircuit is solved on iterations $k-1$ and k, respectively. If the window is $[t_a, t_b]$, then the subcircuit can be iteration latent in subinterval $[t_a, t_x]$, for some $t_x \leqslant t_b$. If $t_x < t_b$, then t_x is the last time point in the window for which the waveforms of I_k match those of I_{k-1} within the specified tolerance. The subcircuit solution of the previous iteration can be used in the interval $[t_a, t_x]$, whereas solution points in the interval $(t_x, t_b]$ must be recomputed on iteration k. The final waveforms for the window are equivalent to those that would have been computed if the I_{k-1} input waveforms were applied to the subcircuit through time t_x, and the I_k input waveforms were applied after t_x. The discontinuity of the effective input waveforms at t_x can cause problems if the tolerance used to compare the input waveforms is too large. Artificial glitches may occur in the output waveforms and excessive time points may be computed after the discontinuity due to the glitches. In extreme cases, one of these discontinuities can lead to a failure of the integration algorithm, as the step size control mechanism repeatedly reduces the step size in a vain attempt to reach an acceptable truncation error computed from divided difference estimates of derivatives of discontinuous functions.

RELAX2.3 uses iteration latency when the subcircuit is latent over the entire window, but not when the subcircuit is latent over only the first part of the window. The likelihood of anomalous behavior due to input waveform discontinuities is reduced in this case, since the number of iteration latency boundaries is greatly reduced, and a subcircuit is more likely to be in a dc steady state at a window boundary, although many window boundaries occur during signal transitions in some of the benchmark circuits. Consequently, the policy of using iteration latency only when the subcircuit is latent over the entire window reduces the likelihood of latency-induced errors, but does not assure that such errors will not arise.

Latency-induced errors can be made as small as desired by decreasing the latency tolerances. However, very small tolerances lead to very little latency exploitation. In effect, the use of iteration latency and the setting of the tolerances present a tradeoff of simulation speed versus reliability and accuracy. The latency algorithms of RELAX2.3 produce acceptably accurate waveforms for all the benchmark circuits, using iteration latency tolerances which are identical to the waveform convergence tolerances.

7.3 Impact of Latency on Speedup

Latency exploitation reduces the number of computations which must be performed, and therefore reduces the uniprocessor run time, τ_1. If some of the eliminated computations are in the critical path of the task graph, then the parallel run time will also be reduced, provided the overhead of latency management is not too large. The parallel completion time of the program on an unlimited number of processors without overhead is given by

$$\tau_\infty = \sum_{x \in \{\text{tasks in } P\}} w_x , \tag{7.1}$$

where w_x is the CPU time of task x, and P is a path in the task graph which maximizes the expression in (7.1). If the overhead for detecting latency is negligible, then w_x for each task x will be reduced or remain the same when latency is exploited. Consequently, latency exploitation will reduce the overall parallel run time, or at worst leave it unchanged, neglecting the overhead required to manage the latency.

The speedup on unlimited processors neglecting overhead is $S_{U,\infty}=\tau_1/\tau_\infty$. Since both τ_1 and τ_∞ are reduced when latency is exploited, the speedup may increase or decrease. If the tasks in P exhibit more latency than other tasks, then the speedup will tend to increase when latency is exploited. Otherwise, the speedup will tend to be reduced. For circuits that partition into subcircuits of nonuniform size, there is a tendency for P to consist of subcircuit evaluation tasks of the larger subcircuits. This is especially true for Gauss-Jacobi, where P typically contains instances of 1 or 2 of the largest subcircuits. Subcircuit evaluation tasks for large subcircuits typically have many input waveforms from other subcircuits. The likelihood that all of the inputs will match the previous iteration, or that all the inputs will be constant in some time interval is small by comparison with smaller subcircuits. This intuitive argument suggests that latency exploitation will tend to reduce speedup, when the number of processors is large, and when the subcircuit partitioning is nonuniform.

As a counter-example, consider a circuit which during a given time window consists of two essentially independent parts, one part consisting of a few small tightly coupled subcircuits and the other consisting of large loosely coupled subcircuits. Since the small subcircuits are tightly coupled, they will required more iterations than the large subcircuits, and the large subcircuits will become iteration latent after a few iterations. In this case, the large subcircuits in the critical path will exhibit greater latency than the average subcircuit, and the speedup may be increased by latency exploitation. Although this situation is less likely than the previous example, it demonstrates that latency exploitation does not necessarily reduce the amount of available parallelism.

The effectiveness of time point pipelining is also affected by latency exploitation. Suppose P is a path in the full window task graph consisting of relatively large subcircuit evaluation tasks $x_1, \ldots, x_4$, and that time points occur at times $t_1, \ldots, t_5$ in each task. The first task cannot be iteration latent, since there is no previous iteration. Subsequent tasks in the path will tend to be iteration latent over increasing portions of the first part of the window. For example, suppose x_2 is nonlatent, x_3 is iteration latent through t_2, and x_4 is iteration latent through t_4. If the latency is not exploited, if 4 processors are available, and if only the tasks in path P are considered, then the parallel execution of time points will proceed as shown in Table 7.1, and will require 8 units of processor time, compared to 20 units of time on a single processor. If partial-window latency is exploited, the parallel execution will proceed as shown in Table 7.2, and will require the same amount of processor time, 8 units. However, the uniprocessor time is reduced to 14 by exploiting the latency. Consequently, in this example, iteration latency exploitation reduces the uniprocessor run time but leaves the 4-processor run

Table 7.1. Parallel Schedule Example: No Latency

Time	x_1	x_2	x_3	x_4
1	t_1			
2	t_2	t_1		
3	t_3	t_2	t_1	
4	t_4	t_3	t_2	t_1
5	t_5	t_4	t_3	t_2
6		t_5	t_4	t_3
7			t_5	t_4
8				t_5

Table 7.2. Parallel Schedule Example: Latency

Time	x_1	x_2	x_3	x_4
1	t_1			
2	t_2	t_1		
3	t_3	t_2		
4	t_4	t_3		
5	t_5	t_4	t_3	
6		t_5	t_4	
7			t_5	
8				t_5

time unchanged.

The parallel execution time will be reduced by latency if one or more of the tasks in P are iteration latent over the entire window. For example, if x_3 is iteration latent over the entire window, then x_4 may compute each nonlatent time point one step earlier. As noted for the full window technique, the critical paths in the task graph tend to contain large tasks which are less likely to be latent over the entire window than other tasks. Therefore, iteration latency exploitation typically reduces the amount of parallelism in time point pipelining.

7.4 FWT Latency Implementation

The window-level exploitation of iteration latency in RELAX2.3 is readily implemented in the FWT program, because the entire input waveforms are available when a subcircuit evaluation task is started. Since FWT uses the augmented task graph $\tilde{T}$, the entire input waveforms of task $(k-1, i)$ as well as the inputs of (k, i) are guaranteed to be available when task (k, i) begins executing. Consequently, iteration latency can be checked for the entire window before computing the first time point of the subcircuit, and if latency is detected, no time points are computed. If the subcircuit is not iteration latent, time latency is checked before computing each time point. When time latency is detected, the input waveforms may be searched immediately for the next change in value, which determines the next time point at which the subcircuit must be evaluated.

The implementation of latency in FWT affects only the subcircuit evaluation task algorithm, which is given below in Algorithm 7.1. The current window boundaries are represented by t_a and t_b, the iteration number is k, and the subcircuit number is i. The convergence and successor checks are the same as in Algorithm 5.7.

Algorithm 7.1. FWT Subcircuit Evaluation Task with Latency (k,i)

```
determine time $t \leqslant t_b$ such that the inputs to $(k,i)$ match the previous
                    iteration on $[t_a, t]$
**if** $(t = t_b)$ $v_i^{(k)}(t) \leftarrow v_i^{(k-1)}(t)$, for all $t \in [t_a, t_b]$
**else** {
    $t\_done \leftarrow t_a$
    pick next time point $t\_next$
    **while** $(t\_done \leqslant t_b)$ {
        **if** $((k,i)$ is in dc steady state at time $t\_done)$ {
            determine time $t \leqslant t_b$ such that the inputs to $(k,i)$
                        are constant on $[t\_done, t]$
            $v_i^{(k)}(t) \leftarrow v_i^{(k)}(t\_done)$
        }
        **else** solve subcircuit at time $t \in (t\_done, t\_next]$
        $t\_done \leftarrow t$
        pick next time point $t\_next$
    }
}
check successors
check convergence
```

7.5 TPPL Latency Implementation

Time point pipelining presents unique problems for the exploitation of time latency and iteration latency. Problems arise because only partial waveforms are generally available when time points are computed, and because long latent intervals can cause pipelining bottlenecks.

When time latency is detected at time t, it may not be possible to immediately determine the next time for which the task should be evaluated, because the input waveforms may not be available through the time of the next input change. Consequently, it may be necessary to mark the task as being time latent and suspend its execution. A mechanism must then be provided for the task to be restarted when one of its inputs changes value. Note that this condition for restarting a time latent task is different from the condition for restarting a nonlatent task. In a nonlatent task, the time of the next time point, *t_next* is determined before the task is suspended, and the predecessors restart the task after they have all progressed through time *t_next*, regardless of whether or not their waveforms change value.

Bottlenecks can occur if time advancements are not propagated through time latent tasks. For example, consider a task x with a set of predecessors P and a set of successors S. Suppose that x is time latent on the interval $[t_\alpha, t_\omega]$, and that the tasks in P and S will all be evaluated at times $t_1, t_2, \cdots, t_m \in (t_\alpha, t_\omega)$. This is not an unreasonable situation, because tasks in S may also be immediate successors of tasks in P, and the waveforms fed from P to S may be active even though those waveforms fed from P to x may be unchanging. Task x does not compute any time points in the interval $(t_\alpha, t_\omega]$ because it is time latent. However, if the output waveforms of x are not updated until the time point is reached at the end of the latent interval, at t_ω, then the tasks in S will not be able to compute the time point at t_1 until after the tasks in P have progressed all the way to time t_ω. This bottleneck can be avoided. If all tasks in P have progressed to time t_1 and x is still time latent, then the waveforms of x can be extended to time t_1 without computing a new solution point, and the successors of x can then be allowed to compute new solutions at time t_1. In terms of the TPP control variables, the bottleneck is avoided by having *t_done* of the latent task track the value of *t_ready*, and by periodically updating *t_ready* to track the progress made by the predecessor tasks during the latent interval.

If time latency is not handled explicitly, and the time steps are determined exclusively based on local truncation error, then time advancements cannot be propagated through latent tasks as described above. Therefore, the explicit detection of time latency has a beneficial influence on parallelism.

The use of iteration latency in time point pipelining presents a more fundamental problem than time latency. The window-level approach to iteration latency used in FWT is not particularly suitable for time point pipelining. Suppose that the window-level latency approach were used for task x with predecessors P and successors S. The determination that x is iteration latent cannot be made until all of x's input waveforms are available over the entire window. In this case the output waveforms of x are not known until all the tasks in P reach the end of the window, which means that all the tasks in S will be delayed until the tasks in P are finished. If latency were not exploited, the time points would propagate through x one at a time, allowing tasks in S to be executing concurrently with tasks in P. Thus, window-level latency exploitation has a serious negative impact on the pipelining parallelism.

Similar bottlenecks appear in the window-level latency approach even when tasks are not iteration latent over the entire window. If task x is iteration latent on $[t_a, t]$, for some $t < t_b$, then the determination that x is not latent on the window cannot be made until all the tasks in P reach time t. Only then can x compute its first time point. If $t >> t_a$, the delay in starting to solve task x will result in a significant decrease in parallelism. This parallelism reduction will be comparatively small in early iterations since the iteration latency boundaries will be close to t_a, but the parallelism reduction will be comparatively large in later iterations as the iteration latency boundaries approach t_b.

By allowing iteration latency to be exploited on partial windows, the time point pipelining bottlenecks described above can be avoided. When partial window latency is used, the latency status of a task can be determined time point by time point as the input waveforms advance. For example, if the input waveforms to task x are available through time t, and they match the previous iteration through time t, then the output waveforms of task x for the current iteration can be copied from the previous iteration through time t and these waveforms can be made available to S. If at time $t_x > t$ one of the inputs to x differs from the previous iteration, then x becomes nonlatent at time t_x and begins computing solution points. Just as in the time latency case, it is important to propagate time advancements through iteration latent tasks to avoid bottlenecks on long latent intervals.

The detection of iteration latency requires comparing the current input waveforms with the input waveforms of the previous iteration. Therefore, when checking iteration latency at time t, it is necessary that the input waveforms of the previous iteration be available through time t. In order to guarantee that the previous iteration waveforms will be available, the augmented task graph $\tilde{T}$ must be used, rather than the unaugmented task graph used in the TPP program without latency. The PARASITE results of Chapter 4 indicate that the penalty for using $\tilde{T}$ is small except when the number of processors is very large.

Time latency and iteration latency have been implemented in an experimental version of the TPP program which bears the unimaginative but functional name TPPL. The algorithm for the subcircuit evaluation task is outlined below in Algorithms 7.2 - 7.6. The TPP control variables retain their meanings as defined in Chapter 6, except the *status* variable can take on additional values to indicate when a task is iteration latent or time latent. When the TPP control variables appear without subscripts, the subscripts k, i are assumed. The value of δ is smaller than the smallest possible time step.

Algorithm 7.2. TPPL Subcircuit Evaluation Task (k, i)

```
more ← TRUE
while (more = TRUE) {
    if (status = UNINITIALIZED) tppl_init (k, i)
    else if (status = EVALUATE) tppl_eval (k, i)
    else if (status = ITER_LATENT) iter_lat (k, i)
    else if (status = TIME_LATENT) time_lat (k, i)
    else if (status = END_ITER_LATENT) end_iter_lat (k, i)
    else if (status = END_TIME_LATENT) end_time_lat (k, i)
    else if (status = CONVERGE_CHECK) {
        if (global convergence achieved and $t_b < t_f$)
                    start next window
        more ← FALSE
    }
    if (more = TRUE) {
        LOCK(lock)
        if (status = EVALUATE) limit t_next
        update t_ready and waiting_for
        if (waiting_for ≠ NULL) more ← FALSE
        UNLOCK(lock)
    }
}
```

Algorithm 7.3. Tppl_eval (k, i)

```
while (status = EVALUATE and t_ready ≥ t_next) {
    if (t_done ≥ $t_b$) status = CONVERGE_CHECK
    else if ((k, i) is in dc steady state)
        determine time t ≤ t_ready such that the inputs to (k, i) are
                    constant on [t_done, t]
        if (t > t_next) {
            $t_1$ ← t_done
            t_done ← t
            t_next ← min{t + δ, $t_b$}
            status ← TIME_LATENT
            check successors
        }
    }
```

```
        if $(status = EVALUATE)$ {
                solve subcircuit at time $t \in (t\_done, t\_next]$
                $t\_done \leftarrow t$
                pick next time point $t\_next$
                check successors
        }
}
```

Algorithm 7.4. Iter_lat (k, i)

```
determine time $t \leqslant t\_ready$ such that the inputs to $(k, i)$ match the previous
                iteration on $[t_a, t]$
if $(t > t\_done)$ {
        $t\_done \leftarrow t$
        $t\_next \leftarrow \min\{t + \delta, t_b\}$
        check successors
}
if $(t\_done = t_b$ or $t\_done < t\_ready)$ $status \leftarrow END\_ITER\_LATENT$
else $t\_next \leftarrow \min\{t + \delta, t_b\}$
```

Algorithm 7.5. Time_lat (k, i)

```
determine time $t \leqslant t\_ready$ such that the inputs to $(k, i)$ are constant
                on $[t\_done, t]$
if $(t > t\_done)$ {
        $t\_done \leftarrow t$
        $t\_next \leftarrow \min\{t + \delta, t_b\}$
        check successors
}
if $(t\_done = t_b$ or $t\_done < t\_ready)$ $status \leftarrow END\_TIME\_LATENT$
```

Algorithm 7.5. End_iter_lat (k, i)

```
$v_i^{(k)}(t) \leftarrow v_i^{(k-1)}(t)$, for all $t \in [t_a, t\_done]$
pick next time point $t\_next$
$status \leftarrow EVALUATE$
```

Algorithm 7.6. End_time_lat (k, i)

```
$v_i^{(k)}(t\_done) \leftarrow v_i^{(k)}(t_1)$
pick next time point $t\_next$
$status \leftarrow EVALUATE$
```

7.6 Results

The effect of iteration and time latency exploitation on a uniprocessor is demonstrated in Table 7.3. The run times using Gauss-Seidel waveform relaxation with and without latency are listed along with the run times of the standard direct method. The latency results were obtained using time latency and window-level iteration latency. Two observations are apparent: latency has a significant impact on the run time, and waveform relaxation is not as fast as the direct method for most of the circuits in the benchmark set. The exception is *digfi*, which is the largest and most uniformly partitioned of the circuits, and has comparatively weak coupling between subcircuits. It should be noted that the benchmark circuits were obtained primarily from real industrial circuits, and they were not prescreened on the basis of suitability for waveform relaxation, aside from the fact that only MOS circuits were considered. Also, the automatic partitioning and windowing algorithms were used without manual optimization. Results have been cited in the literature which demonstrate that waveform relaxation is capable of speeds greater than the direct method on a uniprocessor for a variety of circuits [Whi86]. In the context of this chapter, the important point demonstrated by Table 7.3 is that the use of latency is important in making the uniprocessor performance of waveform relaxation to be as competitive as possible with alternative

Table 7.3. Uniprocessor Run Times (in seconds)

Circuit	Direct Method	WR-GS	WR-GS with Latency
dvs	57	127	97
dpla	54	110	65
scdac	420	1029	568
ben2k	297	594	331
digfi	1342	1934	742

algorithms.

As previously noted, the effect of time latency in waveform relaxation is small because waveform relaxation is intrinsically a multirate integration algorithm. The impacts of iteration and time latency are shown separately in Table 7.4, where it is seen that the time latency effect is negligible for the benchmark circuits. By contrast, iteration latency accounts for a speed improvement close to 2 on a uniprocessor.

The results of the FWT program with latency enabled are given in Tables 7.5 and 7.6. The reference times in computing the speedups are the Gauss-Seidel uniprocessor run times with latency exploitation. As expected, the speedups are generally less than those obtained without latency. This, of course, does not mean that the run times are longer when latency is used, it only means that the contribution of parallel processing is reduced somewhat.

The performance of the TPPL program is summarized in Tables 7.7 and 7.8. The speedups are computed with respect to the FWT program with latency running on 1 processor. Since TPPL exploits iteration latency over partial windows and FWT exploits iteration latency only over entire windows, TPPL exhibits a speed improvement over FWT even on 1 processor in some cases. In cases marked with asterisks, the

Table 7.4. Uniprocessor Speedups Due to Latency: Gauss-Seidel

Circuit	No Latency	Iteration Latency	Time Latency	Both Latencies
dvs	1.0	1.39	1.00	1.37
dpla	1.0	1.74	1.02	1.81
scdac	1.0	1.84	0.99	1.82
ben2k	1.0	1.70	0.99	1.79
digfi	1.0	2.51	0.98	2.58

Table 7.5. FWT Gauss-Seidel Speedups with Latency

Circuit	Processors			
	1	2	4	8
dvs	1.0	1.4	1.3	1.3
dpla	1.0	1.7	2.1	1.9
scdac	1.0	1.9	3.3	4.0
ben2k	1.0	1.6	1.8	1.8
digfi	1.0	1.9	2.9	3.1

Table 7.6. FWT Gauss-Jacobi Speedups with Latency, Normalized to Gauss-Seidel

Circuit	Processors			
	1	2	4	8
dvs	0.7	1.3	2.0	2.6
dpla	0.7	1.3	2.4	3.3
scdac	0.6	1.2	2.1	3.2
ben2k	0.5	1.0	1.9	2.7
digfi	0.6	1.2	2.1	3.4

Table 7.7. TPPL Gauss-Seidel Speedups

Circuit	Processors	
	1	8
dvs	1.1	2.4
dpla	1.1	3.5
scdac	1.2	7.8
ben2k	0.6*	1.6*
digfi	1.2	5.2

Table 7.8. TPPL Gauss-Jacobi Speedups, Normalized to Gauss-Seidel

Circuit	Processors	
	1	8
dvs	0.6	2.9
dpla	0.6	3.4
scdac	0.6	4.1
ben2k	0.2*	1.3*
digfi	0.6	3.3*

tolerances had to be reduced below the default values. For the *ben2k* circuit, the iteration latency tolerance, local truncation error tolerance, and Newton convergence tolerance were all reduced in order to obtain accurate waveforms, and this resulted in a significant degradation in the run times. For the *digfi* run marked with an asterisk, the iteration latency tolerance had to be reduced to avoid a failure of the integration algorithm at an effective input waveform discontinuity. In the cases which did not experience such difficulties, the TPPL performance on 8 processors surpassed the FWT performance on 8 processors.

The Gauss-Seidel TPPL speedups compare favorably with the results excluding latency in Table 6.1. The superior speedups are due to the use of partial window iteration latency, rather than to an increase in parallelism. The Gauss-Jacobi speedups with latency are significantly less than those obtained without latency. In the Gauss-Jacobi case, the effect demonstrated in Table 7.2 is especially applicable, since a critical path in an augmented Gauss-Jacobi task graph normally consists of instances of the largest subcircuit, which is unlikely to be latent over the entire window. A critical path in a Gauss-Seidel task graph normally contains a mixture of different subcircuits, some of which may be latent and others which are not. Consequently, when Gauss-Jacobi time point pipelining is used for a circuit which is dominated by a large subcircuit, latency does not have a significant effect on the parallel execution time if the number of processors is large compared to the available parallelism. However, the parallel run time is reduced in those cases where the number of processors is small compared to the available parallelism, due to the reduction in the number of computations. This can be observed more clearly in Table 7.9 where the run times of Gauss-Jacobi time point pipelining on 8 processors show little improvement due to latency for the small circuits *dvs* and *dpla*, but show a significant improvement for *digfi*, where the subcircuit partitioning is more uniform and where the number of processors is small compared to the available parallelism.

Table 7.9. Transient Analysis Run Times

Method	Procs.	Circuit				
		dvs	dpla	scdac	ben2k	digfi
Direct	1	57	54	420	297	1342
FWT-GS	8	91	55	209	278	501
FWT-GJ	8	40	27	218	150	403
TPP-GS	8	54	39	173	218	419
TPP-GJ	8	32	24	218	160	433
FWT-L-GS	8	70	33	121	168	216
FWT-L-GJ	8	34	25	162	105	201
TPP-L-GS	8	40	19	73	210*	140
TPP-L-GJ	8	31	19	138	257*	226*
Key: L=latency exploited; *=tighter tolerances used						

Finally, Table 7.9 shows the run times of the different parallel waveform relaxation methods on 8 processors, and the run times of the direct method on 1 processor. Parallel waveform relaxation on 8 processors is faster than the direct method on 1 processor, even if latency is not exploited and even though these circuits do not have good waveform relaxation performance on 1 processor. Further performance improvements are attainable with latency exploitation on 8 processors, but this performance advantage comes at the expense of reduced reliability.

CHAPTER 8

CONCLUSIONS

Four different parallel forms of the basic waveform relaxation algorithm with windowing have been studied. These parallel algorithms use Gauss-Seidel or Gauss-Jacobi relaxation in combination with either the full window technique or the time point pipelining technique for coordinating the parallel execution of computations. The superiority of Gauss-Seidel over Gauss-Jacobi for the solution of MOS digital circuits on a uniprocessor is well known. The use of Gauss-Jacobi relaxation for circuit simulation has been largely avoided in previous work, because more iterations are required for convergence. However, when sufficiently many processors are available, the overall run time of Gauss-Jacobi is less than that of Gauss-Seidel. This relationship between Gauss-Seidel and Gauss-Jacobi has been established formally for the linear algebraic equation case in a theorem which relates the spectral radii of the Gauss-Seidel and Gauss-Jacobi iteration matrices to the relative degrees of available parallelism. Results of the PARASITE parallel simulation time estimator, which produces accurate estimates of the parallel run times neglecting overhead, shows that the extra parallelism of Gauss-Jacobi waveform relaxation is more than sufficient to result in faster run times than Gauss-Seidel, when the number of processors is sufficiently large. These results are confirmed by actual multiprocessor circuit simulations, using the FWT and TPP parallel waveform relaxation programs running on an Alliant FX/8. For 4 of the 5 benchmark circuits, using the full window technique, the performance of the Gauss-Jacobi method surpasses that of Gauss-Seidel on 8 processors. For the other circuit, PARASITE indicates that Gauss-Jacobi will be faster than Gauss-Seidel on 16 or more processors.

Time point pipelining exposes more parallelism than the full window technique, and it introduces additional overhead. The results of the FWT and TPP programs confirm that time point pipelining produces faster run times than the full window technique in those cases where the full window technique does not expose enough parallelism to keep the processors busy. The previously unexplored combination of time point pipelining and the Gauss-Jacobi method has been shown to be the fastest of the four basic parallel waveform relaxation algorithms when the number of processors is large enough. PARASITE estimates indicate that speedups of about one order of magnitude should be possible on about 32 processors for 1000-transistor circuits, where the speedup is computed with respect to the Gauss-Seidel method on a single processor.

The available parallelism of the Gauss-Seidel and Gauss-Jacobi methods can be estimated prior to simulating a circuit, by performing a computationally inexpensive analysis of the task graphs. A presimulation selection procedure has been presented which uses these estimates to choose either the Gauss-Seidel or Gauss-Jacobi method for simulating a given circuit on a given number of processors. Furthermore, by using the parallelism estimates to predict the potential processor utilization, the selection procedure chooses either the full window technique or time point pipelining.

The uniprocessor speed of the basic waveform relaxation algorithm is improved significantly when iteration latency (or partial waveform convergence) is exploited. On parallel processors, iteration latency exploitation reduces the amount of parallelism in the benchmark circuit examples, but the overall performance is still better when latency is exploited. Since latency exploitation reduces the parallelism to a greater extent for the algorithms with greater parallelism, the differences in run times of the different algorithms are reduced when latency is exploited. The choice of the latency tolerances involves a tradeoff between greater latency exploitation and reliability, since loose tolerances can lead to inaccurate waveforms or failures of the integration algorithm. The reliability problems are most severe when time point level iteration latency is used.

Waveform relaxation on a uniprocessor works best for circuits which can be partitioned into a large number of weakly coupled subcircuits of nearly the same size. These are the same circuit properties which lead to good speedups in parallel waveform relaxation. Even though some of the benchmark circuits used in obtaining the experimental results do not satisfy this criterion, the parallel waveform relaxation run times on 8 processors are significantly less than the uniprocessor run times of both waveform relaxation and the standard direct methods. Further speed improvements are possible from the natural parallelism of waveform relaxation if more processors are used. When large subcircuits arise due to large subsets of tightly coupled nodes, the same techniques used to parallelize the standard direct methods can be used within the large subcircuits to expose levels of parallelism which go beyond the natural relaxation parallelism studied in this thesis.

REFERENCES

[All85] *FX/FORTRAN Programmer's Handbook*, Alliant Computer Systems Corp., Acton, Mass., May 1985.

[All86a] *FX/Series Architecture Manual*, Alliant Computer Systems Corp., Acton, Mass., Jan. 1986.

[All86b] *Concentrix C Handbook*, Alliant Computer Systems Corp., Acton, Mass., Aug. 1986.

[Bau78] G. Baudet, "Asynchronous Iterative Methods for Multiprocessors," *J. of the ACM*, vol. 25, no. 2, pp. 226-224, April 1978.

[Bis86] G. Bischoff and S. Greenberg, "CAYENNE: A Parallel Implementation of the Circuit Simulator SPICE," *IEEE Int. Conf. on Computer-Aided Design*, pp. 182-185, Santa Clara, Calif., Nov. 1986.

[Bry84] R. E. Bryant, "A Switch-Level Model and Simulator for MOS Digital Circuits," *IEEE Trans. Computers*, vol. C-28, pp. 178-183, March 1979.

[Cal79] D. A. Calahan and W. G. Ames, "Vector Processors: Models and Applications," *IEEE Trans. Circuits and Systems*, vol. CAS-26, pp. 715-726, Sept. 1979.

[Cal80] D. A. Calahan, "Multi-Level Vectorized Sparse Solution of LSI Circuits," *IEEE Int. Conf. on Circuits and Computers*, pp. 976-979, Oct. 1980.

[Cha69] D. Chazan and W. Miranker, "Chaotic Relaxation," *Linear Algebra and Its Applications*, vol. 2, pp. 199-222, 1969.

[Chu75] L. O. Chua and P-M Lin, *Computer-Aided Analysis of Electronic Circuits*, Prentice-Hall, Englewood Cliffs, 1975.

[Cox87] P. Cox, R. Burch, D. Hocevar, and P. Yang, "SUPPLE: Simulator Utilizing Parallel Processing and Latency Exploitation," *IEEE Int. Conf. on Computer-Aided Design*, pp. 368-371, Santa Clara, Calif., Nov. 1987.

[Deu84] J. T. Deutsch and A. R. Newton, "MSPLICE: A Multiprocessor-Based Circuit Simulator," *Int. Conf. Parallel Processing*, pp. 207-214, May 1984.

[Dum87] D. Dumlugöl, P. Odent, J. P. Cockx, and H. J. De Man, "Switch-Electrical Segmented Waveform Relaxation for Digital MOS VLSI and Its Acceleration on Parallel Computers," *IEEE Trans. Computer-Aided Design*, vol. CAD-6, pp. 992-1005, Nov. 1987.

[Fly72] M. J. Flynn, "Some Computer Organizations and Their Effectiveness," *IEEE Trans. Computers*, vol. C-21, pp. 948-960, Sept. 1972.

[Gal88] K. A. Gallivan, P. Koss, S. Lo, and R. A. Saleh, "A Comparison of Parallel Relaxation-Based Circuit Simulation Techniques," *Electro'88*, Boston, Mass., May 1988.

[Gan59] F. R. Gantmacher, *Applications of the Theory of Matrices*, Interscience Publishers, 1959.

[Gar79] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W. H. Freeman and Co., New York, 1979.

[Gea71] C. W. Gear, *Numerical Initial Value Problems in Ordinary Differential Equations*, Prentice-Hall, Englewood Cliffs, New Jersey, 1971.

[Haj83] I. N. Hajj and D. G. Saab, "Symbolic Logic Simulation of MOS Circuits," *IEEE Int. Symp. on Circuits and Systems*, pp. 246-249, Newport Beach, Calif., May 1983.

[Ham83] S. D. Hamm and S. R. Beckerich, "VAMOS: Circuit Simulation Program for a Vector Computer," *IEEE Int. Conf. on Computer-Aided Design*, pp. 252-253, Santa Clara, Calif., 1983.

[Hsi85] H. Y. Hsieh, A. E. Ruehli, and P. Ledak, "Progress on Toggle: A Waveform Relaxation VLSI-MOSFET CAD Program," *IEEE Int. Symp. on Circuits and Systems*, pp. 213-216, Kyoto, Japan, June 1985.

[Hwa84] K. Hwang and F. A. Briggs, *Computer Architecture and Parallel Processing*, McGraw Hill, New York, 1984.

[Jac87] G. K. Jacob, A. R. Newton, and D. O. Pederson, "Parallel Linear-Equation Solution in Direct-Method Circuit Simulators," *IEEE Int. Symp. on Circuits and Systems*, pp. 1056-1059, Philadelphia, Pennsylvania, May 1987.

[Kan85] S. M. Kang, "Circuit Simulation for CMOS VLSI," *IEEE Int. Symp. on Circuits and Systems*, pp. 907-980, Kyoto, Japan, June 1985.

[Kuc86] D. J. Kuck, E. S. Davidson, D. H. Lawrie, and A. H. Sameh, "Parallel Supercomputing Today and the Cedar Approach," *Science*, vol. 231, pp. 967-974, Feb. 28, 1986.

[Lel82] E. Lelarasmee, A. E. Ruehli, and A. L. Sangiovanni-Vincentelli, "The Waveform Relaxation Method for Time-Domain Analysis of Large-Scale Integrated Circuits," *IEEE Trans. Computer-Aided Design*, vol. CAD-1, pp. 131-145, July 1982.

[Mat86] S. Mattisson, "CONCISE A Concurrent Circuit Simulation Program," Ph.D. dissertation, Dept. of Appl. Electronics, Univ. of Lund, Sweden, Aug. 1986.

[May83] "A Device Clustering Algorithm for Vectorized Circuit Simulation," *IEEE Int. Symp. on Circuits and Systems*, pp. 238-241, May 1983.

[Mok85] M. E. Mokari-Bolhassan, D. Smart, and T. N. Trick, "A New Robust Relaxation Technique for VLSI Circuit Simulation," *IEEE Int. Conf. on Computer-Aided Design*, pp. 26-28, Santa Clara, Calif., Nov. 1985.

[Nag75] L. W. Nagel, "SPICE2: A Computer Program to Simulate Semiconductor Circuits," Electronics Research Laboratory Report, ERL M520, Univ. of Calif., Berkeley, May 1975.

[Nak87] T. Nakata, N. Tanabe, H. Onozuka, T. Kurobe, and N. Koike, "A Multiprocessor System for Modular Circuit Simulation," *IEEE Int. Conf. on Computer-Aided Design*, pp. 364-367, Santa Clara, Calif., Nov. 1987.

[New84] A. R. Newton and A. L. Sangiovanni-Vincentelli, "Relaxation-Based Electrical Simulation," *IEEE Trans. Computer-Aided Design*, vol. CAD-3, pp. 308-330, 1984.

[Ort70] J. M. Ortega and W. C. Rheinboldt, *Iterative Solution of Nonlinear Equations in Several Variables*, Academic Press, New York, 1970.

[Rab76] N. B. Rabbat and H. Y. Hsieh, "A Latent Macromodular Approach to Large-Scale Sparse Networks," *IEEE Trans. Circuits and Systems*, vol. CAS-23, pp. 745-752, Dec. 1976.

[Rao85] V. B. Rao and T. N. Trick, "Switch-Level Timing Simulation of MOS VLSI Circuits," *IEEE Int. Symp. on Circuits and Systems*, pp. 229-232, Kyoto, Japan, June 1985.

[Rue87] A. Ruehli, ed., *Circuit Analysis, Simulation and Design*, North-Holland Pub., Amsterdam, 1987.

[Sad87] P. Sadayappan and V. Visvanathan, "Circuit Simulation on a Multiprocessor," *Custom Integrated Circuits Conf.*, pp. 124-128, Portland, Oregon, May 1987.

[Sak81] K. A. Sakallah, "Mixed Simulation of Electronic Integrated Circuits," Ph.D. dissertation, Carnegie-Mellon Univ., Nov. 1981.

[Sal87a] R. A. Saleh, "Nonlinear Relaxation Algorithms for Circuit Simulation," Electronics Research Laboratory Report UCB/ERL M87/21, Univ. of Calif., Berkeley, April 1987.

[Sal87b] R. A. Saleh, D. Webber, E. Xia, and A. Sangiovanni-Vincentelli, "Parallel Waveform Newton Algorithms for Circuit Simulation," *IEEE Int. Conf. on Computer Design: VLSI in Computers and Processors*, pp. 660-663, Rye Brook, New York, Oct. 1987.

[Sma87a] D. Smart, "Parallelism in Direct Method Circuit Simulation," Research Report RC-13399, IBM Watson Research Center, Yorktown Heights, New York, 1987.

[Sma87b] D. W. Smart and T. N. Trick, "Increasing Parallelism in Multiprocessor Waveform Relaxation," *IEEE Int. Conf. on Computer-Aided Design*, pp. 360-363, Santa Clara, Calif., Nov. 1987.

[Sma88a] D. Smart and J. White, "Reducing the Parallel Solution Time of Sparse Circuit Matrices Using Reordered Gaussian Elimination and Relaxation," *IEEE Int. Symp. on Circuits and Systems*, Espoo, Finland, June 1988.

[Sma88b] D. W. Smart and T. N. Trick, "Waveform Relaxation on Parallel Processors," to appear in *Int. J. Circuit Theory and Applications*.

[Szy72] S. A. Szygenda, "TEGAS2—Anatomy of a General Purpose Test Generation and Simulation System for Digital Logic," *ACM Design Automation Workshop*, June 1972.

[Uno85] H. Uno, H. Kinoshita, S. Kumagai, I. Shirakawa, and S. Kodamma, "A Parallel Implementation of MOS Digital Circuit Simulation," *IEEE Int. Conf. on Computer-Aided Design*, pp. 2-4, Nov. 1985.

[Var62] R. Varga, *Matrix Iterative Analysis*, Prentice Hall, Englewood Cliffs, New Jersey, 1962.

[Vla82] A. Vladimirescu, "LSI Circuit Simulation on Vector Computers," Electronics Research Laboratory Report, UCB/ERL M82/75, Univ. of Calif., Berkeley, Oct. 1982.

[Web87] D. M. Webber and A. Sangiovanni-Vincentelli, "Circuit Simulation on the Connection Machine," *24th ACM/IEEE Design Automation Conf.*, pp. 108-113, Miami, Florida, June 1987.

[Wee73] 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 Trans. Circuit Theory*, pp. 628-634, Nov. 1973.

[Whi85a] J. White and A. L. Sangiovanni-Vincentelli, "Partitioning Algorithms and Parallel Implementations of Waveform Relaxation Algorithms for Circuit Simulation", *IEEE Int. Symp. on Circuits and Systems*, pp. 221-224, Kyoto, Japan, June 1985.

[Whi85b] J. White, R. Saleh, A. Sangiovanni-Vincentelli, and A. R. Newton, "Accelerating Relaxation Algorithms for Circuit Simulation Using Waveform Newton, Iterative Step Size Refinement, and Parallel Techniques," *IEEE Int. Conf. on Computer-Aided Design*, pp. 5-7, Santa Clara, Calif., Nov. 1985.

[Whi86] J. K. White and A. Sangiovanni-Vincentelli, *Relaxation Techniques for the Simulation of VLSI Circuits*, Kluwer Pub., Boston, 1986.

[Win80] O. Wing and J. W. Huang, "A Computation Model of Parallel Solution of Linear Equations," *IEEE Trans. Computers*, pp. 632-638, July 1980.

[Yam85] F. Yamamoto and S. Takahashi, "Vectorized LU Decomposition Algorithms for Large-Scale Circuit Simulation," *IEEE Trans. Computer-Aided Design*, vol. CAD-4, pp. 232-239, July 1985.

[Yan80] P. Yang, I. N. Hajj, and T. N. Trick, "SLATE: A Circuit Simulation Program with Latency Exploitation and Node Tearing," *IEEE Int. Conf. on Circuits and Computers*, pp. 353-355, Oct. 1980.

VITA

David Smart was born in Chicago, Illinois, on June 30, 1954. He attended the University of Illinois at Urbana-Champaign and received the B.S. degree in Electrical Engineering in January 1976. He was awarded the General Electric Fellowship in Electrical Engineering to continue his studies at the University, and he received the M.S. degree in May 1977. From 1977 to 1984 he was a member of technical staff in the circuit simulation and analysis group at GTE Communication Systems in Northlake, Illinois. He returned to the University of Illinois, where he was a research assistant at the Coordinated Science Laboratory from 1984 to 1988 while working on the Ph.D. degree. He was also a teaching assistant in the Department of Electrical and Computer Engineering in the spring semester of 1987. In the summer of 1987 he worked at the IBM Thomas J. Watson Research Center in Yorktown, New York. His current technical interests include the simulation, analysis, and design of circuits, computer-aided design of integrated circuits, and applications of parallel processing to engineering and scientific problems.