# Reducing the Parallel Solution Time of Sparse Circuit Matrices Using Reordered Gaussian Elimination and Relaxation

David Smart
Coordinated Science Laboratory
University of Illinois at Urbana-Champaign

Jacob White
Dept. of Electrical Engineering and Computer Science
Massachusetts Institute of Technology

**Abstract**

Using parallel processors to reduce the execution times of classical circuit simulation programs like SPICE and ASTAP has been the focus of much current research. In these efforts, good parallel speed increases have been achieved for linearized system construction, but it has been difficult to get good parallel speed increases for sparse matrix solution. In this paper we examine two approaches for reducing parallel sparse matrix solution time: the first based on pivot ordering algorithms for Gaussian elimination, and the second based on relaxation algorithms. In the section on Gaussian elimination sparse matrix solution, we present a pivot ordering algorithm which increases the parallelism of Gaussian elimination compared to the commonly used Markowitz method. The performance of the new algorithm is compared to other suggested ordering algorithms for a collection of circuit examples. The minimum number of parallel steps for the solution of a tridiagonal matrix is derived, and it is shown that this optimum is nearly achieved by the ordering heuristics which attempt to maximize parallelism. In the section on relaxation, we present an optimality result about Gauss-Jacobi over Gauss-Seidel relaxation on parallel processors.

## 1 Introduction

Designers of high performance integrated circuits make extensive use of circuit simulation programs like SPICE and ASTAP [NAG, WEE] in order to tune their designs before fabrication. These circuit simulation programs often require hours or days to complete a single simulation because they use computationally expensive, but very reliable, numerical techniques. Since many simulations are performed to design a given integrated circuit, slow simulator turn-around time can significantly increase overall design time. For this reason, using parallel processors to reduce the execution times of circuit simulation programs has been the focus of much current research[COX, JAC].

Programs like SPICE and ASTAP use implicit multistep integration algorithms to convert the differential equation system to a sequence of algebraic problems, one for each integration timestep. The algebraic problems are solved using an iterative Newton method, each step of which involves linearizing the circuit about some guessed solution, and solving the generated sparse linear system. Good parallel speed increases have been achieved for the linearized system construction, but not for the sparse linear system solution.

In this paper we examine two approaches for reducing parallel sparse matrix solution time: the first, presented in the following section, based on modified pivot ordering algorithms for Gaussian elimination, and the second, presented in section 3, based on relaxation algorithms. Finally, in Section 4, we present our conclusions and acknowledgements.

## 2 Reordering to Reduce Parallel Solution Time

In this section, the parallel triangularization, or LU factorization, phase of sparse matrix solution is investigated. We begin in the next section by describing the LU factorization algorithm and our computational model. We then present a modification of the commonly used Markowitz pivot ordering algorithm[MAR] which reduces the parallel LU factorization time, and compare it with several other suggested reordering algorithms for a collection of circuit examples. We then derive the optimum parallel solution for a tridiagonal matrix, and show that this optimum is nearly achieved for all pivot reordering algorithms, except Markowitz and some closely related algorithms.

### 2.1 Parallelism in Sparse Factorization

The solution of $Ax = b$, where $A \in \Re^{n \times n}$ and $x, b \in \Re^n$, by Gaussian elimination can be accomplished in three steps: Triangularization or LU factorization, forward elimination, and backward substitution[GOL]. LU factorization is the most time-consuming of the three, and we will concentrate exclusively on it. The LU factorization algorithm is, for dense matrices, given by the following nested loop.

**LU Factorization**
for $k = 1$ to $n-1$ *process pivot k*
  for $i = k+1$ to $n$
    $a_{ik} = a_{ik}/a_{kk}$ *divide*
    for $j = k+1$ to $n$
      $a_{ij} = a_{ij} - a_{ik}a_{kj}$ *update the row entry*

For a given pivot $k$ in LU factorization, all the divide operations can be done concurrently, and once they are completed, all the update operations can be done concurrently. Therefore, for large dense matrices the number of concurrent divide operations is of order $n$, and the number of concurrent update operations is of order $n^2$, at least during the early stages of the algorithm.

Circuit matrices, which are very sparse, have few nonzero entries in their rows and columns, especially in the early stages of the decomposition, before many fill-ins are generated. Therefore, concurrent processing of the divides and then the updates for a single pivot does not provide much parallelism. In order to get greater degrees of parallelism, different pivots must be processed concurrently. The extent to which this is possible depends on the matrix structure, which, in turn, determines the dependencies between operations.

In order to investigate the parallelism available in sparse LU decomposition we use a task graph model[WIN]. It is assumed that each divide and each update operation takes one unit of time, and each such operation is represented in the task graph as one vertex. Dependencies between the operations (e.g. that the divide $a_{ik} = a_{ik}/a_{kk}$ must occur before the update $a_{ij} = a_{ij} - a_{ik}a_{kj}$ ) are represented as directed edges. Note that such a graph for a sparse matrix can only be constructed once the matrix fill-in pattern is known, and therefore the graph topology is a function of the pivot ordering in the LU factorization. The ordering of update operations also affects the graph topology, since different pivots can contribute terms to the same update destination, and these update operations must be performed serially.

Based on this task graph model, and the Markowitz pivoting order, it has been shown that the degree of parallelism for solving circuit matrices can be as high as 10% of the number of rows in the matrix. This suggests that for large circuits, large speedups may be possible, but for medium-sized problems additional parallelism must be exploited. Speedups achieved in practice are also limited by the number of available processors and by overhead due to task management and communications[YAM, COX].



ISCAS'88

## 2.2 Pivot Ordering for Improved Parallelism

For uniprocessor applications the goal of a pivot ordering algorithm is to minimize the total number of matrix operations, and this is realized by using reordering algorithms that minimize fill-in in the matrix. The commonly used Markowitz algorithm produces a near-optimal ordering by, at each step of the elimination, choosing as the next pivot a diagonal entry of the uneliminated submatrix which has the lowest Markowitz count, defined as the product of the number of nonzero column entries and nonzero row entries.

For a parallel processor, the goal of an ordering algorithm should be to reduce the time to complete the decomposition, and this may not necessarily be the solution with the fewest number of operations. One approach to measuring the quality of an ordering algorithm for a parallel processor is to compute the length of the longest directed path in the task graph, which is referred to as the task graph depth. This depth is equal to the minimum number of steps required to compute the LU decomposition given sufficient processors, and ignoring overhead factors such as data communications and task scheduling. Several ordering algorithms [BET, HUA, ZHO] have been proposed which attempt to minimize the depth of the task graph without significantly increasing the total number of matrix operations. These algorithms require a method for monitoring the growth of the task graph depth while choosing pivots to be eliminated.

We propose another ordering algorithm which also attempts to minimize the task graph depth and keep the total number of operations from increasing much. The basic idea is that at a step in the elimination process, a set of candidate diagonal pivots is constructed from those diagonal pivots with low Markowitz counts. From the set of candidate pivots, a large independent set is extracted, that is, a set for which if two pivots $i$ and $j$ are in the set $a_{ij}$ and $a_{ji}$ are both zero. All the pivots in an independent set can be processed concurrently with no conflicts, except that more than one pivot may contribute a term to the same update destination. The algorithm uses an integer parameter $a > 0$ which can be tuned empirically for best performance.

**Large Independent Set (LIS) Reordering**
Repeat until the elimination is completed
  $S = \{\ \}$
  $mcount$= minimum Markowitz count of remaining pivots
  For $d = mcount$ to $mcount + a$
    For each pivot $v$ of Markowitz count $d$
      If $S + v$ is an independent set then $S = S + v$
  Eliminate using the pivots in the set $S$, creating fill-ins

The resulting task graph depths for the LIS ordering algorithm and several other ordering techniques applied to a collection of matrices are given in Table 1. Each column corresponds to a circuit matrix, except P1000 and P2047 are 1000 node and 2047 node tridiagonal matrices. Note that the minimum depth/minimum degree[BET] algorithm frequently produces the smallest depth of the ordering methods, however it tends to significantly increase the total number of operations, making it less desirable when only a limited number of processors is available or when overhead factors are taken into account. The peculiar structure of the ALU circuit produces unusual results. For this circuit the Markowitz and minimum degree/minimum depth orderings produce excessive fill-ins which are avoided in the other orderings. For the other circuits, the orderings which try to minimize the task graph depths result in depths up to about 50% less than the Markowitz method.

## 2.3 The Question of Optimality

Since the heuristic ordering algorithms are not guaranteed to find the globally minimum depth ordering, we are faced with the question of how close the resulting depths are to the minimum possible depth. For a circuit of any significant size, there are too many possible orderings to try them all. Random searches for optimum orderings, nested dissection, and simulated annealing did not produce orderings that were any better than the ones found by the heuristics above.

Table 1. Task graph depths.

| | PLA | OPAMP | RAM | RAM2 | P1000 | ALU | P2047 |
|---|---|---|---|---|---|---|---|
| m | 35 | 32 | 66 | 129 | 1998 | 387 | 4092 |
| a | 24 | 26 | 45 | 98 | 1000 | 264 | 2047 |
| b | 18 | 28 | 30 | 125 | 26 | 25 | 30 |
| z1 | 26 | 29 | 49 | 91 | 1000 | 52 | 2047 |
| z2 | 19 | 28 | 36 | 84 | 29 | 27 | 32 |
| w | 23 | 26 | 42 | 79 | 30 | 30 | 33 |
| l | 19 | 28 | 43 | 91 | 27 | 27 | 30 |
| w* | 19 | 24 | 39 | 75 | 28 | 27 | 31 |
| l* | 19 | 28 | 42 | 75 | 26 | 26 | 30 |
| o | | | | | 23 | | 26 |

Key: m=Markowitz, a=min degree/min depth[BET], b=min depth/min degree[BET], z1,z2=Zhou stages 1 and 2[ZHO], w=Wing/Huang pivot ordering[HUA], l=LIS with a=2, *=optimum update ordering [HUA] applied to the pivot ordering, o=globally optimum pivot and update ordering (when it can be computed).

The quality of the orderings produced by the heuristics can be measured if analytic techniques can be used to find meaningful lower bounds on the task graph depth. It is possible to prove a useful result for tridiagonal matrices[SMA]. Tridiagonal matrices are relatively easy to analyze because, regardless of the pivot order, after the elimination of a pivot the resulting uneliminated portion of the matrix can be reorganized into a tridiagonal matrix of lower order.

**Theorem 1** *For any $d > 0$ , the largest $n$ such that the depth of the task graph associated with a tridiagonal matrix of size $n$ will be less than or equal to $d$ is given by*

$$2^{\lfloor d/3 \rfloor + 1} - 1 + \sum_{j=\lfloor d/3 \rfloor + 1}^{\lfloor d/2 \rfloor} \sum_{i=0}^{d-2j} \binom{j}{i}. \qquad (1)$$

The minimum possible task graph depths for two tridiagonal matrices were computed based on (1), and these optimum depths are given in Table 1. The small differences between the optimum results and the results for the LIS algorithm are due to the fact that LIS tries to choose as many pivots as possible at each stage and does not consider conflicts that will result later in adding terms to common update locations. The poor performance of the Markowitz and minimum degree-based methods for tridiagonal matrices is due to the fact that there are many pivots which can be processed in parallel but are not selected by the algorithm because they do not have minimum degree.

# 3 Parallel Sparse Matrix Solution by Relaxation

Another approach to solving $Ax = b$ is to solve each $i^{th}$ row equation for $x_i$, moving the other unknowns to the right hand side by replacing them with guessed values. The values for the unknowns so approximated can be used to improve the guessed values used in each row computation, with the hope of improving the approximation. This procedure can be repeated, until the approximations stop improving significantly. Algorithms of this form are referred to as relaxation algorithms, and are not commonly used in general circuit simulation programs because the approximations don't always approach the correct solution. However, when applied to the specific problem of the transient analysis of MOS integrated circuits, relaxation algorithms do converge reliably (with some "tuning"). In addition, relaxation algorithms are more easily parallelized, and this has renewed interest in them[SAL, WHI, DUE].

When constructing a relaxation iterative procedure, one has two choices about how to update the unknowns. Either one can first solve all the row equations with some existing set of values for the unknowns, and then replace all the unknowns with the improved approximations; or as one can solve a particular row and immediately replace the associated unknown before solving the next row. The element update equation for the former approach, referred to as Gauss-Jacobi (GJ),

can be written compactly as

$$x_i^{k+1} = \frac{1}{a_{ii}} \left[ b_i - \sum_{j=1}^{j=i-1} a_{ij} x_j^k - \sum_{j=i+1}^{j=n} a_{ij} x_j^k \right], \quad (2)$$

and the latter approach, referred to as Gauss-Seidel (GS), can be written as

$$x_i^{k+1} = \frac{1}{a_{ii}} \left[ b_i - \sum_{j=1}^{j=i-1} a_{ij} x_j^{k+1} - \sum_{j=i+1}^{j=n} a_{ij} x_j^k \right], \quad (3)$$

where $k$ is the iteration index.

When relaxation methods are applied to the matrices associated with the transient simulation of MOS circuits, GS converges much faster than GJ, because the GS relaxation can be ordered to follow the strong directionality of such circuits. In general, one would expect GS to be faster than GJ as each row solution uses more recent information, and this conclusion is supported by the Stein-Rosenberg theory[VAR]. Some parallel relaxation-based MOS circuit simulators use the GS method in an attempt to take advantage of its faster convergence speed[SAL, DUE, WHI]. However, experimental evidence indicates, that when sufficiently many processors are used, GJ results in faster solutions due to its higher degree of parallelism[SMA2, MAT, WEB].

In the next few sections we show that the empirical evidence about the superiority of GJ on parallel processors is supported by a parallel theory analogous to the Stein-Rosenberg result. In the next subsection we present a simplified parallel computation model in order to reexamine the comparison between GS and GJ. We then demonstrate that applied to sparse matrices, GS relaxation has substantial parallelism. Finally, in the last subsection, we present an optimality result for GJ over GS on parallel processors.

## 3.1 A Simplified Parallel Computational Model

The unknown update equations for GJ and GS, Eqns. (2) and (3), 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 this summation, but it will be the same for both GS and GJ. The difference between GS and GJ that effects how much total parallelism can be exploited is that in the case of GJ all the unknowns can be updated simultaneously, and in the case of GS, if $A$ is full, only one unknown can be updated at a time.

In order to more easily examine the difference between the two methods, we will treat (2) and (3) as atomic operations which can be computed in one processor step. In this notation, and assuming sufficiently many processors, one iteration of GJ takes one step, as all the unknown updates can be performed simultaneously, and GS takes $n$ steps if A is full, as the $i^{th}$ unknown must be updated before the $i+1$ update equation can be completed.

## 3.2 Exploiting Sparsity in Parallel GS

It is possible to exploit the sparsity of circuit matrices to increase the parallelism of GS and reduce the number of steps needed to complete an iteration to well below $n$. For example, if $a_{i+1,i} = 0$, then $x_i$ can be updated simultaneously with $x_{i+1}$. It is easy to calculate exactly how many steps it will take to compute a GS iteration on a sparse matrix by examining the following graph constructed from $A$. Let the graph have $n$ nodes, labeled 1 through $n$. For each $a_{ji} \neq 0$, $i < j$, place a directed arc from node $i$ to node $j$ in the graph. The result is an acyclic directed graph, and the depth of this graph is precisely equal to the number of steps required to complete one iteration of GS on a parallel processor.

It is also possible to exploit more parallelism in the GS algorithm by beginning the $k+1^{th}$ iteration before completing the $k^{th}$. For example, if $a_{1j}$ through $a_{1n}$ 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. Combining this technique of overlapping iterations with simultaneously updating independent $x_i$ 's , as mentioned above, yields a parallel GS algorithm which is best characterized by the average number of processor steps between GS iteration completions. We will refer to this number of steps as $r$. It should be noted that $r$ depends only on the nonzero structure of $A$ and for general sparse matrices $r$ can be much less than $n$. For example, if $A$ is tridiagonal, $r = 2$. With this in mind, the winner in a comparison between GJ and GS on parallel processors is less clear, and is the subject of the next section.

## 3.3 The Optimality of Parallel GJ over GS

In order to compare the GJ and GS procedures, we will consider their asymptotic convergence properties. In particular, let $A = L + D + U$ where $L$, $D$, and $U$ are strictly lower triangular, diagonal, and strictly upper triangular matrices, respectively. Define the GJ and GS iteration matrices $M_{GJ} = -D^{-1}(L+U)$ and $M_{GS} = -(L+D)^{-1}U$. It is well known that the asymptotic rates of convergence of GJ and GS are related to $\rho(M_{GJ})$ and $\rho(M_{GS})$ respectively[VAR], where $\rho$ denotes spectral radius. Since $m$ iterations of parallel GJ require $m$ steps and $m$ iterations of parallel GS require $r \cdot m$ steps, in the limit as $m \rightarrow \infty$, it follows that parallel GJ is asymptotically faster than parallel GS if $\rho(M_{GJ})^r$ is less than $\rho(M_{GS})$. For a large class of matrices it can be proved that this is the case, and therefore **parallel GJ is asymptotically faster than parallel GS**. The result is precisely stated in the following theorem:

**Theorem 2** *If the elements of $M_{GJ}$ are nonnegative, and $\rho(M_{GJ}) < 1$, then $\rho(M_{GJ})^r \leq \rho(M_{GS})$.*

Examining nonnegative convergent iteration matrices follows the Stein-Rosenberg approach to the GS/GJ comparison[VAR]. It is also true that the matrices associated with the transient simulation of MOS circuits, if the discretization timestep is small[WHI], satisfy the conditions of the theorem, which suggests the comparison plays a direct role in practice.

The proof of Theorem 2 utilizes the following lemma[GAN] and definition:

**Lemma 1** *Perron-Frobenius: If matrix $M \in \Re^{n \times n}$ is nonnegative, then it has a nonnegative real eigenvalue equal to its spectral radius and a nonnegative eigenvector associated with that eigenvalue.*

**Definition 1** *Define $x^{PGJl}, x^{PGSl} \in \Re^n$ to be the vectors of most recently updated elements produced by $l$ processor steps of parallel GJ and GS algorithms respectively.*

Since a GJ iteration finishes in one processor step, $x^{PGJl} = x^l$, the $l^{th}$ iterate of a GJ iteration, and therefore

$$x_i^{PGJl+1} = \frac{b_i}{a_{ii}} + \sum_{j \neq i, j=1}^{j=n} -\frac{a_{ij}}{a_{ii}} x_j^{PGJl}. \quad (4)$$

In general, $x^{PGSl}$ never corresponds to any $x^k$ iterate of GS, because overlapping of the iterations implies that the elements of $x$ most recently updated can correspond to different iterations. And because of data dependencies inherent in the GS algorithm, many of the elements of $x^{PGSl}$ are the same as those of $x^{PGSl+1}$. Therefore, each GS update will be given by either

$$x_i^{PGSl+1} = x_i^{PGSl} \quad (5)$$

or

$$x_i^{PGSl+1} = \frac{b_i}{a_{ii}} + \sum_{j \neq i, j=1}^{j=n} -\frac{a_{ij}}{a_{ii}} x_j^{PGSm(j)}, \quad (6)$$

where $m(j) \in \{0, ..., l\}$. Note that $m(j)$ is used to indicate that in order to follow the GS update formula, it may be necessary to pick out elements from several different, but earlier, $x^{PGSm(j)}$ vectors.

In order to complete the proof, we now consider solving $Ax = 0$ by relaxation, where $A$ is such that $M_{GJ}$ exists and is nonnegative and $\rho(M_{GJ}) < 1$. Let the initial guess $x^0 = x^{PGS0} = x^{PGJ0}$ be a nonnegative eigenvector associated with a nonnegative eigenvalue of $M_{GJ}$ equal to $\rho(M_{GJ})$. Since $M_{GJ}$ is nonnegative, $a_{ij}/a_{ii} \leq 0$ for all $i \neq j$.

Consequently, since $x^0$ is nonnegative and $b = 0$, each term in (4) and (6) is nonnegative for all $i, l$. Also, since $x^0$ is an eigenvector associated with an eigenvalue equal to $\rho(M_{GJ}) < 1$, $x^{PGJl} = \rho(M_{GJ})^l x^0$ and therefore $x_i^{PGJl}$ decays monotonically with $l$ for all $i$.

Suppose $\rho(M_{GJ})^r > \rho(M_{GS})$, then in the limit as $l \to \infty$ the nonnegative vector $x^{PGSl}$ will be less than $x^{PGJl}$. We will show by induction that this can not happen, and this will complete the proof. First, assume $x_i^{PGJl} \leq x_i^{PGSm}$ for all $i$ and all $m \in \{0, ..., l\}$. This assumption clearly holds for $l = 0$, thus forming the basis of the induction. In those cases where (5) applies, $x_i^{PGSl+1} = x_i^{PGSl} \geq x_i^{PGJl}$, and $x_i^{PGSl+1} \geq x_i^{PGJl+1}$ because $x_i^{PGJl}$ is monotone decreasing in $l$. In those cases where (6) applies, each term of the summation in (4) is less than or equal to the corresponding term in (6) and all the terms are nonnegative. Hence $x_i^{PGSl+1} \geq x_i^{PGJl+1}$. Since $x_i^{PGJl}$ decreases monotonically, $x_i^{PGJl+1} \leq x_i^{PGJm} \leq x_i^{PGSm}$ for all $m < l$. Consequently, $x_i^{PGJl+1} \leq x_i^{PGSm}$ for all $i$ and all $m \in \{0, ..., l+1\}$, thus completing the induction and the proof.

# 4 Conclusions and Acknowledgements

Matrix reordering heuristics for parallel processing have been shown to be effective in reducing the depth of the LU factorization task graph. However, for most of the circuit examples investigated the improvement in depth was less than 50% , and the benefit of this improvement is only realizable on a large number of processors. One reason for this is that the Markowitz algorithm which was used as the basis for comparison tends to produce small task graph depths simply by keeping the total number of operations small. For certain matrix structures, the heuristics which attempt to maximize parallelism produce vastly superior results, such as for tridiagonal matrices. In the examples that were considered, the LIS algorithm consistently produced good orderings.

In the examination of relaxation methods we proved a result that indicates on a parallel processor GJ relaxation is almost certainly superior to GS, almost regardless of the numerical character of the problem. The result, we think, is also interesting because it associates the spectral radius of a matrix to some of its graphical, rather than numerical, properties. For example, Theorem 2 leads to the somewhat surprising conclusion that if a tridiagonal matrix satisfies the conditions of the theorem, then the the spectral radius of its associated GS iteration matrix is no less than half the spectral radius of its associated GJ iteration matrix.

The authors would like to acknowledge the many valuable discussions with A. Sangiovanni-Vincentelli, Tim Trick, Don Webber, Res Saleh, Vish Visvanathan, and Vasant Rao. In addition, we would like to mention the paper by Chazan and Miranker[CHA], from which some of the ideas for the analysis were derived. This research was funded in part by Sandia National Laboratory contract 02-8522, Semiconductor Research Corporation contract 86-12-109, and by the Defense Advanced Research Projects Agency under contract N00014-87-K-0825.

# References

[BET] R. Betancourt, "Efficient parallel processing technique for inverting matrices with random sparsity," *IEE Proc.*, pp. 235-240, July 1986.

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

[COX] P. Cox, R. Burch, B Epler, "Circuit Partitioning for Parallel Processing," *IEEE Int. Conf. on Computer-Aided Design*, pp. 186-189, Nov. 1986.

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

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

[GOL] G. Golub, C. F. Van Loan, *Matrix Computations*, The Johns Hopkins University Press, Baltimore, Maryland, 1983.

[HUA] J. W. Huang, O. Wing, "Optimal Parallel Triangulation of a Sparse Matrix," *IEEE Trans. on Circuits and Systems*, pp. 726-732, Sept. 1979.

[JAC] G. Jacobs, D. Pederson, "An Empirical Analysis of the Performance of a Multiprocessor-based Circuit Simulator," *Proc. of the Design Automation Conference*, Las Vegas, Nevada, June 1986.

[MAR] H. M. Markowitz, "The Elimination Form of the Inverse and Its Application to Linear Programming," *Management Science*, pp. 225-269, 1957.

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

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

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

[SMA] D. Smart, "Parallelism in Direct Method Circuit Simulation," Research Report RC-13399, IBM Watson Research Center, Yorktown Heights, New York, 1988.

[SMA2] D. Smart, T. Trick, "Increasing Parallelism in Multiprocessor Waveform Relaxation," *IEEE Int. Conf. on Computer-Aided Design*, pp. 360-363, Nov. 1987.

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

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

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

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

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

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

[ZHO] V. Zhou, "Optimal Parallel Triangulation of a Sparse Matrix–A Graphical Approach", *IEEE Int. Symp. Circuits and Systems*, pp. 624-627, 1981.