A Parallel Band-Matrix Solver for a Circuit Simulation Problem

Luís Miguel Silveira

Research Laboratory of Electronics

Department of Electrical Engineering and Computer Science

Massachusetts Institute of Technology

Abstract

This report sumarizes and evaluates an implementation of a general band-matrix solver algorithm on a massively parallel multiprocessor, the "Connection Machine". The band algorithm is used in a nonlinear relaxation-based circuit simulator for solving the system of equations associated with the node voltages, at each Newton iteration at each time step.

Acknowledgments

This report is part of the requisites for course 18.435/6.848, taught by Prof. F. Thomson Leighton at the Massachusetts Institute of Technology in the Fall Term of 1988.

I thank Andrew Lumsdaine who wrote the original sequential version of the simulator and did most of the circuit element models for the "Connection Machine" version. Toghether, we worked out the data structures we used and defined the global simulation control structure. Without his help and cooperation this work would not have been made.

I acknowledge the cooperation of Prof. Leighton over all phases of the implementation by providing important suggestions for the general data structures, and that of Prof. Jacob White for providing me with some inside into the simulation problem at hand, and assisting in all stages of the project.

I further acknowledge the support of "Thinking Machines Corporation" and its staff of "Connection Machine" developpers for all the help given during this first phase of the project.

Contents

1 Introduction: The Circuit Simulation Problem

Circuit simulation tools, like SPICE and ASTAP [NAG, WEE]. are extensively used nowadays by designers of high performance integrated circuits in order to tune their designs before fabrication.

These circuit simulation programs use very reliable, numerical techniques which allow them to obtain very accurate results as to the performance of the circuits being fabricated. However, these techniques are computationally very expensive and since many simulations may have to be performed in order to ascertain the correct design for a given integrated circuit, the overall design time can be significantly increased by slow simulation turnaround time.

For this reason, much of the current research in the field of circuit simulation has been geared towards using parallel processors, in order to reduce the execution time of circuit simulation programs [JAC, COX, SAL, SIL].

Programs like SPICE and ASTAP use what are generally called Direct Method techniques. They convert the differential equation system to a sequence of algebraic problems, one for each integration timestep, by using an implicit multistep integration algorithm. An iterative Newton method, is then applied to solve the algebraic problems, which involves linearizing the circuit about some guessed solution, and solving the generated sparse linear system. Constructing the system is an easily parallelizable task and good speed increases have been achieved on that problem, but there does not seem to be an efficient way of solving the sparse linear system, particularly if there are many (more than 128) processors available.

If one wants to use machines with many processors, an effective technique for MOS circuits is to use Gauss-Jacobi relaxation. Since the Gauss-Jacobi algorithm turns the problem of calculating the solution of a system of equations into that of calculating the solution to each equation independently for its associated unknown and then passing around the computed values, Gauss-Jacobi can be tailored to exploit as many processors as there are equations to be solved. Although Gauss-Jacobi is guaranteed to converge under certain conditions, can nevertheless be very inefficient if the timestep required to achieve convergence becomes very small. Gauss-Jacobi has another very undesirable feature, that it can converge to a false solution, which makes it somewhat unreliable.

In order to improve the reliability of Gauss-Jacobi while still maintaining a very parallel algorithm an algorithm was devised, that extracts a variable-sized diagonal band of the matrix and solves that band directly [LUM]. This leads to a relaxation method, Band Relaxation, with better convergence properties but still very parallelizable.

This section and the next one, follow very nearly the description and motivation for the Band-Relaxation Simulation as presented in [LUM].

The next section will address in somewhat more detail the question of applying relaxation techniques to the general circuit simulation problem. The band-relaxation concept is presented and in some sense compared to

Gauss-Jacobi. Section 3 describes the algorithm that was used to solve the band system described in section 2, and a study of its sequential and parallel complexity are presented. Section 4 addresses the general framework of parallel circuit simulation related to this implementation. Afterwards, in section 5, the parallel band-solver implementation is described and results are presented in section 6. Finally some conclusions are drawn and some directions for further work on the subject are also dicussed.

2 Nonlinear Band-Relaxation Circuit Simulation

Under very broad assumptions, using nodal formulation, the equations associated to the circuit to be simulated are: equations:

$$\dot{q} + f(v, u) = 0 \tag{1}$$

where q, the vector of the charges in the circuit nodes, is a function of node voltages, and $f(u, v)$, the vector of node conductances, a function of the node voltages v and the input sources u.

By using some multistep integration algorithm, the above equations are changed into a sequence of algebraic problems, one for each integration timestep. For instance, by applying the Implicit-Euler Method

$$x(n+1) = x(n) + h \;\; f(u(n+1), v(n+1)) \tag{2}$$

we get the equations

$$\dot{q}(v(n+1)) + h \;\; f(v(n+1), u(n+1)) - \dot{q}(v(n)) = 0 \tag{3}$$

An iterative Newton method, is then applied to solve, at each time step, the algebraic problems, which involves linearizing the circuit about some guessed solution.

In Nonlinear Band-Relaxation, each iteration of the iterative Newton method involves computing the solution to a sparse linear system of the form $Ax = b$, where $A \in \Re^{n \times n}$ is the discretized Jacobian of the nonlinear system, $x \in \Re^n$ is the vector of node voltage updates, and $b \in \Re^n$ is the Newton residue. The solution of this system is obtained in most circuit simulators by means of some sparse Gaussian elimination technique, which is difficult to parallelize [WIN].

As stated before, the iterative relaxation algorithms are more easily parallelized methods and can also produce a solution to the given system. Of all the methods, the simplest is, as mentioned before, the Gauss-Jacobi relaxation method. Using Gauss-Jacobi to determine the solution of the system implies solving each equation independently for its associated unknown, and then exchanging the computed values, repeating the process until a consistent solution throughout is achieved. The element update equation for Gauss-Jacobi can be written as

$$x_i^{k+1} = \frac{1}{a_{ii}} \left[b_i \; - \sum_{j=1, j \neq i}^{n} a_{ij} x_j^k \right], \tag{4}$$

where k is the iteration index and a_{ij} is the ij^{th} entry of the matrix A (the Jacobian of the circuit equations at each time step).

It is obvious that Gauss-Jacobi can easily exploit n processors effectively, as the equations represented by the rows in A are solved independently. Gauss-Jacobi is also guaranteed to converge when applied to solving the matrices generated by the transient analysis of MOS circuits, if the timestep is small enough and there is a capacitor to ground at each node. The problem with Gauss-Jacobi is that it is inefficient since the timestep required to achieve convergence may be very small, and because false convergence is common due to the limited spread of information with each iteration [SAL,WEB,WHI].

To better understand why band-relaxation can provide a better overall result, while keeping a high degree of parallelism, we can expand the notion exploited by the Gauss-Jacobi method, by noting that any two matrices $M, N \in \Re^{n \times n}$ are said to be a splitting of the matrix A if $A = M - N$. The iteration equation for a relaxation algorithm based on a given splitting is then

$$Mx^{k+1} = Nx^k + b, \tag{5}$$

and the asymptotic rate of convergence is the spectral radius of $(M^{-1}N)$ [VAR]. It becomes clear that if the splitting is chosen such that M is easy to factor and the convergence of (5) is somewhat faster, we get a better method for solving the system of equations. Note that M should be easy to factor in a parallel fashion, since the objective is to produce an efficient parallel algorithm.

Based on what was said, Gauss-Jacobi is equivalent to choosing a diagonal M, which is already in factored form. As we said, this leads to a very efficient parallel algorithm, but presents poor convergence properties. One would expect that movement of off-diagonal elements of A from N to M would accelerate the convergence, because more of the system is being solved directly. This conjecture can be proven for the case of an A which is diagonally dominant with positive diagonals and negative off-diagonals [VAR]. If M is selected to be a band matrix with a band-size much less than n, then there are methods for solving the system which present some parallelism and can provide us with better overall results. The method discussed in the next section is one of them. The relaxation algorithm generated by using a banded M will be referred to as "Band Relaxation".

I will leave out of this discussion all the problems associated with choosing a band-size and selecting elements for a band. For our purposes here, it is enough to say that there are several ways of doing that, but the subject relating to which elements to place on the band is still an open question in some sense. Basically, and taking into account the physical significance of the matrix we are splitting, we would like to keep in the band all the off-diagonal elements that correspond to tightly coupled connections in the circuit. By using such a criteria, we expect, and results seem to prove that, to reliabily improve the covergence properties of the algorithm.

The present version of the simulator does some reordering of the equations based on a given coupling criteria [LUM], therefore constructing a graph of tightly coupled components, and uses a version of the Reverse-Cuthill-McKee (RCM) algorithm [DUF,GEO] to reorder that graph in order to minimize the bandwith of A. The really important issue here is the coupling criteria, whose choice is still an open research subject. The actual coupling criteria used is

$$\frac{\max^2(a_{ji}, a_{ij})}{|a_{ii}a_{jj}|} \geq \alpha \tag{6}$$

where α is a given constant whose value will somehow determine the band-size. Using the RCM algorithm on the graph built based on the above criteria, will reorder the equations in order to minimize the bandwidth of A, therefore increasing the available parallelism, without loosing speed of convergence.

The folowing is a pseudo-code description of the overall band-relaxation algorithm.

Algorithm 1 (BR) Given $x, b \in \Re^n$ and $A, M \in \Re^{n \times n}$, M being the band of tightly coupled elements contained in A, this algorithm solves $Ax = b$ iteratively for x.

```
band-LUdecomp(M)
x^0 := 0
For k = 1, 2, ...
    For i = 1, 2, ..., N
        sum := 0
        For j = 1, 2, ..., N
            If j < i - p or j > i + p
                sum := sum + a_ij x_j^k
        b'_i := b_i - sum
    band-solve M x^{k+1} = b'
    If x^{k+1} converged with x^k, break
    If k >  max-iter, signal failure
```

The focus of the work related here is the solution of the system

$$Mx^{k+1} = b'$$

The following section describes a parallel method for solving this problem.

3 An Algorithm for Solving Band-Matrices

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

$$Mx^{k+1} = b'$$

where $x, b' \in \Re^n$ and $M \in \Re^{n \times n}$, M being a band matrix of bandwidth m.

The algorithm described in this section is a generalization for block matrices of the algorithm described in [WAN]. The general form of the algorithm is based on a partition of the matrix, perhaps augmenting its size, into P blocks of size k.

Internally the whole matrix is divided in such a way that each $m \times m$ set of rows is in fact a group of three matrices, the **a** matrix composed on the $m \times m$ elements on the m-block diagonal, the subdiagonal **c** matrix and the superdiagonal **b** matrix. Below is a representation for a $5m \times 5m$ matrix (Each **a**, **b**, and **c** are $m \times m$ matrices).

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

Fig. 1 - Matrix divided into 4 partitions with 4 row-blocks each, showing the internal partition in block-matrices **a**, **b** and **c**.

This representation makes it easier to understand the algorithm that follows. The underlying goal is to eliminate the sub- and superdiagonal blocks

b's and **c**'s in order to get a block diagonal matrix.

Below is a graphical interpretation of the 5 stages of the algorithm for a matrix with $P = 4$ and $k = 4$ (see Fig. 2). Each element in the matrix should be considered as a block matrix $\in \Re^{m \times m}$, either a **a**, a **b** or a **c** block. The folowing steps are applied:

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

- For each of the P partitions use the $k-1$-th block-row to eliminate the superdiagonal entry of the block row above it. Do likewise, in a sequential fashion for block-rows $k-2, \cdots, 2, 1$ of all partitions. These operations cause a fill-in to be produced in the last column of each diagonal block and in the last column of each superdiagonal block (see Fig. 4).

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

- Use the last block-row of the last partition to eliminate the superdiagonal fill-in of the last block row of the previous partition. Do likewise in a sequential fashion for partitions $P-1, \cdots, 2$ (see Fig. 7). Afterwards, for each of the P partitions, use the last block-row of each partition, to eliminate the superdiagonal fill-in of the first block-row of that partition. Do likewise for block-rows $2, 3, \cdots, k-1$ of that partition (see Fig. 8).

- Solve the $m \times m$ block diagonal systems.

$$
\left|\begin{array}{cccc|cccc|cccc|cccc}
\hline
x & x & & & & & & & & & & & & & & \\
x & x & x & & & & & & & & & & & & & \\
 & x & x & x & & & & & & & & & & & & \\
 & & x & x & x & & & & & & & & & & & \\
\hline
 & & & x & x & x & & & & & & & & & & \\
 & & & & x & x & x & & & & & & & & & \\
 & & & & & x & x & x & & & & & & & & \\
 & & & & & & x & x & x & & & & & & & \\
\hline
 & & & & & & & x & x & x & & & & & & \\
 & & & & & & & & x & x & x & & & & & \\
 & & & & & & & & & x & x & x & & & & \\
 & & & & & & & & & & x & x & x & & & \\
\hline
 & & & & & & & & & & & x & x & x & & \\
 & & & & & & & & & & & & x & x & x & \\
 & & & & & & & & & & & & & x & x & x \\
 & & & & & & & & & & & & & & x & x \\
\hline
\end{array}\right|
$$

Fig. 2 - Matrix divided into 4 partitions with 4 row-blocks each

$$
\begin{array}{|cccc|cccc|cccc|cccc|}
\hline
x & x & & & & & & & & & & & & & & \\
& x & x & & & & & & & & & & & & & \\
& & x & x & & & & & & & & & & & & \\
& & & x & x & & & & & & & & & & & \\
\hline
& & & x & x & x & & & & & & & & & & \\
& & & x & & x & x & & & & & & & & & \\
& & & x & & & x & x & & & & & & & & \\
& & & x & & & & x & x & & & & & & & \\
\hline
& & & & & & & x & x & x & & & & & & \\
& & & & & & & x & & x & x & & & & & \\
& & & & & & & x & & & x & x & & & & \\
& & & & & & & x & & & & x & x & & & \\
\hline
& & & & & & & & & & & x & x & x & & \\
& & & & & & & & & & & x & & x & x & \\
& & & & & & & & & & & x & & & x & x \\
& & & & & & & & & & & x & & & & x \\
\hline
\end{array}
$$

Fig. 3 - Matrix after the first step of the algorithm

$$
\left|\begin{array}{cccc|cccc|cccc|cccc}
\hline
x & & & x & & & & & & & & & & & & \\
 & x & & x & & & & & & & & & & & & \\
 & & x & x & & & & & & & & & & & & \\
 & & & x & & & & x & & & & & & & & \\
\hline
 & & & x & x & & & x & & & & & & & & \\
 & & & x & & x & & x & & & & & & & & \\
 & & & x & & & x & x & & & & & & & & \\
 & & & x & & & & x & & & & x & & & & \\
\hline
 & & & & & & & x & x & & & x & & & & \\
 & & & & & & & x & & x & & x & & & & \\
 & & & & & & & x & & & x & x & & & & \\
 & & & & & & & x & & & & x & & & & x \\
\hline
 & & & & & & & & & & & x & x & & & x \\
 & & & & & & & & & & & x & & x & & x \\
 & & & & & & & & & & & x & & & x & x \\
 & & & & & & & & & & & x & & & & x \\
\hline
\end{array}\right|
$$

Fig. 4 - Matrix after the second step of the algorithm

$$
\begin{array}{|cccc|cccc|cccc|cccc|}
\hline
x & & & x & & & & & & & & & & & & \\
 & x & & x & & & & & & & & & & & & \\
 & & x & x & & & & & & & & & & & & \\
 & & & x & & & & x & & & & & & & & \\
\hline
 & & & x & x & & & x & & & & & & & & \\
 & & & x & & x & & x & & & & & & & & \\
 & & & x & & & x & x & & & & & & & & \\
 & & & & & & & x & & & & x & & & & \\
\hline
 & & & & & & & x & x & & & x & & & & \\
 & & & & & & & x & & x & & x & & & & \\
 & & & & & & & x & & & x & x & & & & \\
 & & & & & & & & & & & x & & & & x \\
\hline
 & & & & & & & & & & & x & x & & & x \\
 & & & & & & & & & & & x & & x & & x \\
 & & & & & & & & & & & x & & & x & x \\
 & & & & & & & & & & & & & & & x \\
\hline
\end{array}
$$

Fig. 5 - Matrix after the first part of step 3

$$
\left|\begin{array}{cccc|cccc|cccc|cccc}
\hline
x & & & x & & & & & & & & & & & & \\
& x & & x & & & & & & & & & & & & \\
& & x & x & & & & & & & & & & & & \\
& & & x & & & & x & & & & & & & & \\
\hline
& & & & x & & & x & & & & & & & & \\
& & & & & x & & x & & & & & & & & \\
& & & & & & x & x & & & & & & & & \\
& & & & & & & x & & & & x & & & & \\
\hline
& & & & & & & & x & & & x & & & & \\
& & & & & & & & & x & & x & & & & \\
& & & & & & & & & & x & x & & & & \\
& & & & & & & & & & & x & & & & x \\
\hline
& & & & & & & & & & & & x & & & x \\
& & & & & & & & & & & & & x & & x \\
& & & & & & & & & & & & & & x & x \\
& & & & & & & & & & & & & & & x \\
\hline
\end{array}\right|
$$

Fig. 6 - Matrix after the third step

$$
\begin{array}{|cccc|cccc|cccc|cccc|}
\hline
x & & & x & & & & & & & & & & & & \\
& x & & x & & & & & & & & & & & & \\
& & x & x & & & & & & & & & & & & \\
& & & x & & & & & & & & & & & & \\
\hline
& & & & x & & & x & & & & & & & & \\
& & & & & x & & x & & & & & & & & \\
& & & & & & x & x & & & & & & & & \\
& & & & & & & x & & & & & & & & \\
\hline
& & & & & & & & x & & & x & & & & \\
& & & & & & & & & x & & x & & & & \\
& & & & & & & & & & x & x & & & & \\
& & & & & & & & & & & x & & & & \\
\hline
& & & & & & & & & & & & x & & & x \\
& & & & & & & & & & & & & x & & x \\
& & & & & & & & & & & & & & x & x \\
& & & & & & & & & & & & & & & x \\
\hline
\end{array}
$$

Fig. 7 - Matrix after the first part of step 4

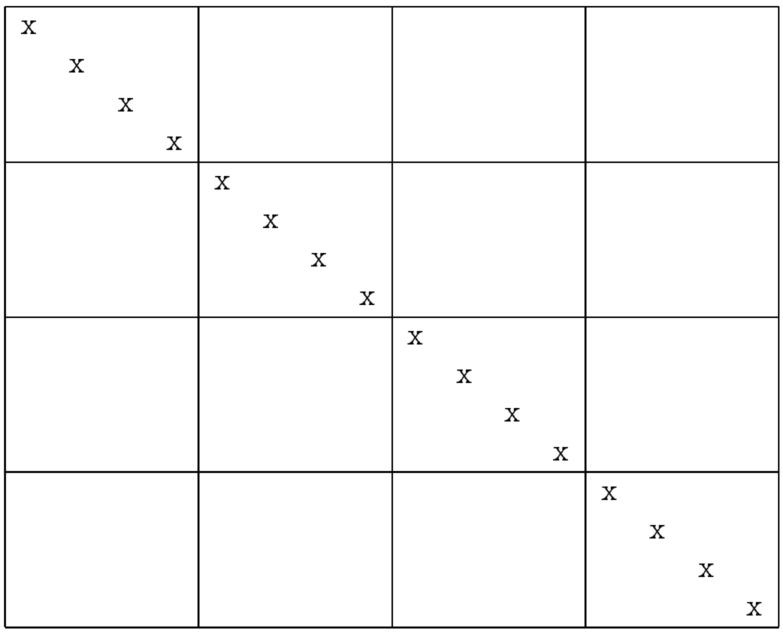

Fig. 8 - Matrix after the fourth step

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

In the next sections, the complexity of this algorithm is analysed, both in its sequential form and in its parallel form.

3.1 Sequential Complexity

In this section the sequential complexity of the modified Wang algorithm is obtained.

In table 1, the five stages of the algorithm are expressed in terms of their complexity. Note that since this version of the algorithm involves block matrices of size m, each step of each stage requires the factorization of an $m \times m$ matrix, the solution of m $m \times m$ systems of equations based on the factorized matrix, up to three $m \times m$ matrix products and additions. So the total number of steps is the one shown below times a constant factor (the sum of all individual steps just seen). However for our purposes, the number of steps involving the above operations is the apropriate mesure.

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

Table 1: Sequential Complexity of the Modified Wang algorithm

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

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

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

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

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

the minimum is found for:

$$\frac{\partial SS}{\partial k} = 0 \quad \Rightarrow k = \sqrt{\frac{N}{m}}$$

So the optimum choices are:

$$\boxed{k = P = \sqrt{\frac{N}{m}}}$$

However, this choice is not of great interest, since the term $5Pk$ which dominates the complexity formula found, is in fact independent of k and P.

To conclude, Wang's algorithm, if implemented on a sequential machine, has a complexity of $O(N)$. This is not surprising, because a carefull look tells us that the algorithm is nothing more than a simple, smart, two-way Gaussian elimination on the band elements. On the next section, however, we'll see how this algorithm exploits available parallelism to solve the system of equations.

3.2 Parallel Complexity

In this section the parallel complexity of the modified Wang algorithm is obtained, according to the implementation performed.

In table 2, the five stages of the algorithm are expressed in terms of their complexity. Note that again, since this version of the algorithm involves block matrices of size m, each step of each stage requires the factorization of an $m \times m$ matrix, the solution of m $m \times m$ systems of equations based on the factorized matrix and, again, up to three $m \times m$ matrix products and additions. So the total number of steps is the one shown below times a constant factor (the sum of all individual steps just seen). However, again, for our purposes, the number of steps involving the above operations is the apropriate mesure.

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

Table 2: Parallel Complexity of the Modified Wang algorithm

Let us once again try to find the values P and k for which the number of parallel steps is minimized. This will tell us of the effective complexity of the algorithm. Remembering that m is determined by physical considerations, we'll look for the relation between P and k such that the two values minimize the total number of steps.

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

$$PS = 2P + 4k - 5 = 2\frac{N}{mk} + 4k - 5$$
$$\frac{\partial PS}{\partial k} = -2\frac{N}{mk^2} + 4$$

the minimum is found for:

$$\frac{\partial PS}{\partial k} = 0 \quad \Rightarrow k = \sqrt{\frac{N}{2m}}$$

So the optimum choices are:

$$\boxed{k = \sqrt{\frac{N}{2m}}}\ \boxed{P = \sqrt{\frac{2N}{m}}}$$

With this values, then the complexity is at best:

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

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

This is by no means the best possible result. In fact there are algorithms whose complexity only grows with $O(\log N)$ [DON][DUF]. The decision to choose this algorithm was simply because it presents the nice feature of requiring few non-nearest neighbour communications. In fact, except for stages

$3a$ and $4a$ all comunications can be done on a nearest neighbour basis. In the results section we will further debate whether this turned out to be a good option.

4 Non-Linear Band-Relaxation on the Connection Machine

The algorithm described and studied in the previous sections was implemented as the core of the Band-Relaxation circuit simulation version on the Connection Machine.

After setting up the data-strutures for the simulator, the implementation of the Band-Solver and the device model equations were done at the same time. A preliminary version of the simulator using Gauss-Jacobi was the testbed for debugging the models code. The Band-Solver was first implemented in a sequential fashion, on a sequential machine and later ported into a Connection Machine front-end, for developmente of the parallel version.

An intermediate version was built that incorporates a switch allowing the user to choose from running it in parallel (attached to a Connection Machine) and sequentially. This program, which is a stand-alone module, actually uses the data structures that the circuit simulator also works upon. This significantly reduced the time necessary to incorporate the band-solver on the simulator and made extra hacking unnecessary. The code for this program is shown on appendix A at the end of this report. An additional switch in that program allows us to generate matrices of any given size, and choose the values of P and k. These matrices are generated randomly and the solution of the system for which they are used is the vector whose i-th component has a value of i.

A final version of the simulator with the Band-Solver in place was patched together from that stand-alone code. That version only contains models for Resistors, Capacitors, Diodes, Voltage and Current Sources. The models for Transistors and other devices were left to a second stage of development.

Since the sequential version of the simulator was written in **C**, that was the language we decided to code the parallel version in. However, due to inconsistent support by **Thinking Machines** of the **C*** software, and following the advice of company's responsible, the coding of the whole simulator and the Band-Solver was done in **C/Paris**. This means thall all the front-end code was left in **C**, altough it had to be significantly changed, and all the bf Connection Machine code was done in **Paris**, the machine's high-level assembly language.

Programming in **Paris** is not the way a programmer really likes to put down his code since, being an assembly language, the programmer now has to go deep into the machine's details, set the heap and stack areas, do memory management routines, stack routines, etc. The level of coding used in **Paris** implies that the programmer is really at the lowest level, having to specify every single "move" of data in the **Connection Machine** processors, keeping track of the size of that data, using a stack to push and pop the context flags that define the current active set of processors, etc.

Using **Paris** doesn't gives the programmer the flexibility of a high-level language, makes the code too long and too hard to follow and debug. For an experienced programmer, however, it presents all the advantages (?) of using assembly-like languages.

In the conclusions section, at the end of this report, this subject is further debated, to the point of deciding whether increasing effort should be directed to continuing this project at this level or whether to wait for the new **C*** compiler, whose release is scheduled sometime next year.

4.1 Data Structures

This section on the data structures briefly describes the structures used to hold the data on the band-solver both on the sequential and parallel version.

The Data Structures used for the Band-Solver were chosen in the context of the Circuit Simulator implementation. This implies that the data structures used were by no means the most efficient if one were simply building a stand-alone Band-Solver.

In fact, one can easily think of many settings for the data that would imply a faster solution of the matrix problem, using the described Modified Wang's Algorithm [LEI]. However, since we were not interested in a stand-alone band-solver, but simply on a core function to be used inside another application, the data-structures used had to comply with other criteria.

The basic algorithm for the implementation of the simulator is briefly described in the next section. For our purposes here, we only need to state that at each simulation time step, the matrix elements have to be filled in with the result of some previous calculations (obviously done over the circuit devices), the matrix has to be solved, and the solution of the system, which is the vector of updated node voltages, has to be later sent over, for new device calculations to be performed.

With this setting in mind, our data-structures were set-up in such a way that all this communication between matrix elements and devices would be somewhat minimized. We decided to use a bidimensional structure, consisting of matrix rows and devices. The matrix elements were placed in rows and N **Connection Machine** processors were dedicated to hold the data for those rows. These N processors were placed in a linear fashion, such that each group of m consecutive processors would represent an $m \times m$ matrix. However incorrect, we will refer to these processors as the node processors. An explanation for that will be given in the next section. The other dimension of the whole structure consisted on the devices.

It was said before, that only Resistors, Capacitors and Sources have been implemented as devices. All these devices share the property of being two terminal devices. However the structure can contain i-terminal devices without any change. Each device connected to a given node, say node k, would be placed along the dimension y connected to the processor holding the data for the $k - th$ row of the matrix. All other nodes that were also terminals for that device would have on their list of devices (therefore on the y-th axis, connected to that processor) a fake device, called fan-out device, that would actually perform as an actual device. When one of the device's terminal is ground, that terminal is obviously ommited (the same thing for any power terminal).

So, we have a two-dimensional structure, with the matrix rows on one dimension and the devices, attached to its corresponding terminal nodes on the other dimension. This bi-dimensional structure, and specially the choice of having each row of the matrix stored in the memory of a single **Connection Machine** processor, is certainly not the best way of solving the problem in terms of load balancing, since when the matrix is being solved, the device processors are idle, and when the devices are performing their computations, the row elements are idle.

However, from the point of vue of solving the matrix, this ensures that we minimize the communication between processors (an issue whose importance will be debated ahead) since now each processor has access to a complete row of the matrix locally. And since most of the communication on the Modified Wang Algorithm is made between clusters of m processors (processors holding an $m \times m$ matrix, therefore m rows) this means that the bulk of communication will be between processors a distance $O(m)$ apart.

The next figure gives a view of how the data structure is actually used. Each of the row or devices elements shown (Node1, $\cdots$, Node4, Dev11, $\cdots$, Dev44, etc) is a **Connection Machine** processor. Each of the Node processors hold the necessary data associated with a row of the full matrix. The device processors hold the necessary data to perform their calculation. The next section gives a little more insight into the simulation control algorithm.

Appendix B, at the end of the report, is a listing of the files "nlm.h" and "structs.h", where all the data structures for the circuit simulator objects are defined. In it, one can find an detailed description of how the memory of each processor is used, depending whether it's a node or a device processor. Appendix C contains the listing of several modules from the parallel version of the circuit simulator. Module "CMbuild.c" contains the code necessary to initialize the **Connection Machine** processors, allocate the necessary heap space to hold the data, and load that data into each processor. Module "CMselect" contain the functions that handle the selection of active processors at each step of the simulation algorithm and the Band-Solver al-

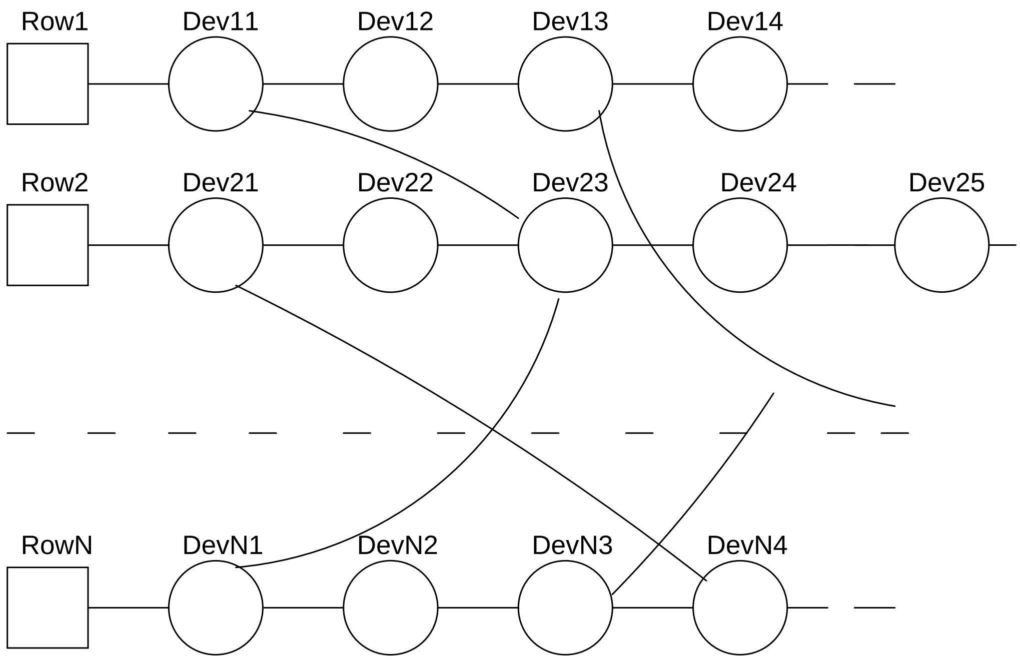

Figure 1: Data-Structures used in the Connection Machine implementation of the Band-Relaxation Circuit Simulator

gorithm, together with the routines that handle the stack memory allocated in each processor. Module "pbcr.c" contains miscellaneous functions for handling processor data and perform a matrix partition. Module "main.c" is the main simulator module. Modules "trap.c", "solve.c" and "nlm.c" contain the code that describes the global simulation control algorithm. Finally modules "res.c", "cap.c" and "vsrc.c" are included for completeness, and are the code used for the calculation of device constituent equations, in the case of resistors, capacitors and voltage sources. All theses modules are intrincate, and are provided only in vue of completeness. As this report does not intend to dwell along the details of the implementation but merely to describe its overall structure, the results obtained and the experience we gained from this work, this code can entirely be ommited.

4.2 Circuit Simulator

The basic control algorithm for the circuit simulator has been layed out in the previous sections. Note that since there are N processor holding the matrix data, one can easily think of each processor being dedicated to a different node in the circuit. This makes it very simple to implement a Gauss-Jacobi like scheme to solve the equations at each time step.

Basically the ideia for the control algorithm used in the simulator stems out of the data structure shown above. At each time step, the front-end determines by some criteria (notably the estimates on the local truncation error) the next time step to be used. Each processor computes a guess for the next voltage step based solely on the last few values it has stored in its memory.

On a first step, that guess is scanned out into the device processors. In the next step each device processor that is not a fan-out ask its fan-outs for the other terminal voltages it needs. Then it computes its associated constituent equations and calculates the device's conductance, capacitance and current. In the next step these values are sent back to the fan-out devices, and in the final step scanned back, along the y-axis, into the correct matrix positions.

Then the Band-Solver is called upon, to solve the band matrix. The final result is the vector of new node voltages, which can then be read in order to determine the status of the convergence of the method.

If one were to do Gauss-Jacobi instead of the Band-Relaxation algorithm, the previous steps would be executed as stated, simply the solution of the system of equations would be done locally, on a equation by equation basis (as characteristic of the Gauss-Jacobi algorithm). This is basically the reason why the processors holding the matrix data are called the node processors.

This flexibility allowed us to implement both methods and establish some comparisons. The results can be seen ahead.

It is not the intention of this report to dwell into the details of the implementation of the Band-Solver, and even less on the Circuit Simulator's implementation. However, some understanding of the global structure is necessary. The folowing is a pseudo-code description of the overall implementation.

Algorithm 2 (PBR)

```
do /* until end of simulation */
    choose the time step to be used
    do /* until convergence */
        do parallel
            guess the new node voltage
            scan the guess value out to the device processors
            get other terminal voltages from the fan-out devices
            use the constituent equations of the device to calculate ca-
              pacitance, conductance and current for the element
            scan values into the processors holding the matrix rows and
              store them in the correct matrix positions
        do parallel
            solve the banded system of equations (for Band-Relaxation)
              and compute the new node voltages) or
            compute the new node voltages locally (if using Gauss-
              Jacobi)
    while x^{k+1} doesn't converge with x^k
while current time < simulation time
```

5 Solving Band-Matrices on the Connection Machine

This section deals in a little more detail with the Band-Solver algorithm, based on the Modified Wang Algorithm and used in the implementation of the Band-Relaxation Circuit Simulator on the **Connection Machine**.

As it was said before, the band algorithm was first implemented as a stand-alone module and only later connected to the circuit simulator modules. This intermediate version was used for code debugging and some data experiments, along with some performance measurements. Its code can be seen in appendix A at the end of the report.

That intermediate version consists of 4 major blocks: the sequential band-solver (module "swang.c"), the parallel band-solver (module "pwang.c"), the main program (module "mwang.c"), the data-structures setting module ("pbuild.c") and a module containing some miscellaneous functions to handles processors stack area ("pselect.c"). The sequential band-solver is a version of the Modified Wang Algorithm that runs on any sequential machine. Its performance is debated on the results section and its complexity was studied in a previous section.

The module "mbuild.c" also contains the routines necessary to allocate the heap memory and the stack memory used by the processors, along with all the routines that manage and keep track of the stack pointer in each processor.

The data-strucures used on the sequential version were chosen to match as closely as possible the structure to be used in the parallel version. Therefore, the matrix is stored as a series of records, each containing the necessary data related to a row of the complete matrix. All operations on the matrix, hence, involve several of these records and the data stored in them.

These records can be thought of as being the sequential equivalents to the row processors of the **Connection Machine**. The only difference being, of course, that there are no operations going in parallel, which is related to the fact that the processors on the **Connection Machine** can be regarded as active and intelligent data structures, but their sequential equivalents, the records, are merely static data to be accessed from a global memory.

The parallel band-solver is the couterpart of the sequential version. Developped for the **Connection Machine**, it uses a very similar global structure. This happens due to the **SIMD** (Single Instruction over Multiple Data) structure of the underlying machine, which makes the parallel code look like a slice of the overall sequential code.

Each row processor (or record in the sequential version) contains six vectors of data. Of these six vectors, three vectors hold the rows of the $m \times m$ matrices **a**, **b** and **c**. Two others hold the rows of the matrices that describe the fill-ins that appear below and above the diagonal (**fil** and **fih**). The sixth, **aux_inv** is used to hold intermediate data that is necessary to keep at each elimination step.

The elimination of an $m \times m$ block matrix, involves the LU decomposition of another matrix, the solution of a system of the form:

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

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

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

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

Since the code, being written in Paris, is by itself not very clear and straightforward, every stage of the algorithm is performed by a different function, that calls the general LU-decomposition routine, then determines the solution of the given m $m \times m$ systems (by backward and forward substitution), and adds the result to the correct place. This is done in three steps. First the pivot matrix is decomposed (note that there are P decompositions going on in parallel !) and left in place. Then the multiple system solution is found, and the result is left in the "aux_inv" matrix in the correct place. Afterwards, a matrix multiplication and addition is done using the "aux_inv" matrix, with all the elements in the row where the element to be eliminated is. For instance one step of the second stage of the algorithm typically consist of the following:

decompose the a_i matrix and keep it in place solve the right system $x^T a_{i_{LU}} = b_{i-1}$ and leave the result in aux_inv

form the products $c_i \;\; aux\_inv_i$, $b_i \;\; aux\_inv_i$, $fil_i \;\; aux\_inv_i$

actualize the i-th block row (a block row is a row of $m \times m$ matrices) by using the subtracting the above products from a_{i-1}, fih_{i-1} (which is created) and fil_{i-1}

At the time the implementation was done, we had no knowledge of the performance difference between a "read" and a "write" into another's processor memory. Due to that, the algorithm was based on reads, rather than on writes, which in vague terms may imply that the results obtained can be decreased by a factor of two. That is nontheless a second order factor in this project, not because the performance issue is not important, which it is, but because other issues turned out to be more important than implementation ones such as this (for instance the algorithm's complexity).

6 Experimental Results

In this section we show some of the results obtained from the implementation of the Band-Solver algorithm and the Band-Relaxation Circuit Simulator.

We will first show the results obtain for the Band-Solver alone and discuss them, and afterwards present and discuss the results obtained with the Circuit Simulator, in terms of performance.

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

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

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

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

The results on table 3 predict a break-even point for the parallel implementation that is far beyond our capacity to test at the moment (due to disk space limitations for storing the matrices !). This break-even point is obviously dependent on m and grows with m. We didn't try to reach break-even point measurements because that would not any add extra knowledge to the problem.

Another meaningfull measurement to be obtained, is the number of communication steps used throughout the algorithm. This can be approximately measured by counting the number of gets on a typical run-time. For the optimal choice examples we got as values the ones presented in table 4.

These values have some meaning in terms of the results obatined. Using a 1 m *second* value for an average communication step and taking into account that a *get* (and therefore a *read*) takes approximately twice as much as a *write*, we get a communication burden along the values shown in table 4. Although these values are somewhat of a worst case possibility, they in fact tell us something about the performance of the machine and the implementation of the algorithm. Interesting enough, the values depicted in the table 4 also grow as $O(\sqrt{N})$ as expected.

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

Example	N	P	k	m	Time (Par) (sec)	Time (Seq) (sec)
opt4950	4950	45	22	5	100.03	18,42
opt1000	1000	20	10	5	43,05	3,55
opt490	490	14	7	5	29,10	1,67
x490	490	7	14	5	42,15	1,68
x1000	1000	25	8	5	43,16	3,5
y1000	1000	10	20	5	64,03	3,62
z1000	1000	20	5	10	223,58	11,15
w1000	1000	20	25	2	6,56	1,03
a4970	4970	71	35	2	12,5	5,23

Table 3: Performance Results for the Band-Solver Algorithm

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

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

simple RC trees with incresing number of nodes and devices.

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

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

The results shown are encouraging in terms of circuit simulation. In fact the parallel version on the **Connection Machine** was able to beat the Gauss-Jacobi sequential version (as expected) and a break-even point for the Band-Relaxation Non-Linear Relaxation was in sight for circuits with a reasonably large number of nodes.

And once more the results followed the expected patterns for the sequential and parallel version ($O(N)$ for sequential and $O(\sqrt{N})$ for the parallel version).

On the overall the results must be considered as encouraging. On one hand they follow the predicted theoretical patterns, and on the other hand, they somehow show that efficient Non-Linear Band-Relaxation for moderately to large circuits is possible in a paralle architecture. Several questions were however left unanswered, and we will try to address some of those in the next section.

[1]Predicted results based on extrapolation of previous values

7 Conclusions and further work

From the results presented in the previous section we cannot infer a decisice answer to the question of whether a massively parallel computer is the best testbed for solving band matrices, and therefore for solving the circuit simulation problem. In Appendix D at the end of report we present the profilings obtained while running the two biggest parallel optimal examples (*opt4950* and *opt1000*). From this profiles of run-time, we can see that up to 20% of the total time is spent on a "wait_until_data_arrives" idle sequence, and up to 75% of the time is spent on **Connection Machine** communications related functions, initialization procedures and security checks. Note that the running time of the version with the profile switch on is greater than that of the version without the profile switch, therefore not very adequate for running time measurements.

The question whether a distributed memory parallel machine is a good architecture for solving the Band-Relaxation problem, or whether bus-based shared memory systems should be tried instead, is still an open question in this, as many other fields of research. Implementations on other parallel machines should be tried in order to ascertain the real burden that intensive communication presents on this problem.

Another imporatnt issue relates to the algorithm being used to solve the Band matrix in parallel. The Modified Wang Algorithm presented in this report and used in the implementation here described is certainly not the best possible option. Other methods, like Block Cyclic Reduction [DUF], present a complexity growing with $O(m^3 + m^3 \log(\frac{N}{m}))$, although they are more communication intensive. Also, other methods already known [DON] and promise complexities in the order of the same order.

It is our belief that a second phase of this project should be an implementation, still based on the **Connection Machine**, and following one of these algorithms. An important issue here is the setting of the data-structures. Since the Band-Solver is in fact the bulk of the Simulator, the data-structures should be set such that they would be optimal for it [LEI], instead of finding a compromise between other issues. For the **Connection Machine**, it remains to be proved if using more processors (therefore exploting more parallelism but also with more communication involved) is a viable alternative.

An important issue at this point is the question of the language to be used on the next phases of the project. Since a lot of effort has been invested in **C** which is anyway the most widely used language on the **CAD**-world, there is no question at that level. The problem arises to whether a second implementation on the **Connection Machine**, using for instance the Block Cyclic Reduction Algorithm, should be attempted in **C/PARIS** or one should wait for the next release of **C***, scheduled for sometime in the beginning of 1989. My personal feeling is that it is worth the waiting period for the **C*** language to be settled, since that period can be used for a carefull setting of the data-structures and the whole simulation algorithm to be performed.

A third phase of the project should be instead based on machines with different architectures, in order to get a better understanding of the spectrum of the available possibilities.

References

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

[DON] J. Dongarra and S. Lennart Johnsson,
"Solving banded systems on a parallel Processor",
Parallel Computing, 5(1&2):219-246, 1987.

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

[GEO] A. George and J. W-H. Liu,
Computer Solution of Large Sparse Positive Definite Systems,
Prentice-Hall, Englewood Cliffs, 1981.

[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.

[LEI] F. Thomson Leighton,
"An Introduction to the Theory of Networks, Parallel Computation and VLSI Design",
To be published.

[LUM] A. Lumsdaine, J. White, D. Webber, A. Sangiovanni-Vincentelli,
"A Band Relaxation Algorithm for Reliable and Parallelizable Circuit Simulation",

Proc. International Conference on Computer Aided Design, Santa Clara, California, November 1988.

[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. Saleh and A. R. Newton,
"An Event-Driven Relaxation-Based Multirate Integration Scheme for Circuit Simulation",
Proc. Int. Symp. on Circuits and Systems, Philadelphia, Pennsylvania, May 1987.

[SIL] Luís Miguel Silveira,
"Multiprocessor Implementation of an event-driven Circuit Simulator: Evaluation of Design Options",
European Projects, Esprit 1058 Report, July 1987

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

[WAN] H. H. Wang,
"A Parallel Method for Tridiagonal Equations",
ACM Transactions on Math. Softw., 1981, 7, pp170-183.

[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.