Solving Large and Sparse Linear Equations in Analog Circuit Simulation on a Cluster of Workstations

M. SCHNEIDER*, U. WEVER† AND Q. ZHENG†

** Technische Universität München, Institut für Informatik, D-80290 München, Germany*
† Siemens AG, Zentralabteilung Forschung und Entwicklung BTSE 43, Otto-Hahn-Ring 6, D-81739 München, Germany

For developing analog circuits at Siemens AG, the method of Harmonic Balance was integrated into the circuit simulator TITAN. The equations of Harmonic Balance are solved by Newton's method. The corresponding Jacobian is a very large sparse block matrix. Hitherto this kind of problem was computed either on a single workstation or on a vector computer. In this paper, we present an algorithm that makes it possible to simulate analog circuits on a cluster of workstations. Since the bulk of data is stored in a distributed way, even very large problems which often cannot be processed on vector computers due to insufficient memory capacity may now be solved.

Received June 1993; revised September 1993

1. INTRODUCTION

Frequency analysis is an important aspect in the design of analog circuits like filters, oscillators, modulator and demodulator circuits. In order to cope with autonomous oscillations and general non-linear effects, a variety of algorithms for non-linear frequency analysis has been developed and published. One of the most general algorithms is Harmonic Balance, which has been integrated into the circuit simulator TITAN at Siemens AG, where it is intensively used for the development of different kinds of analog circuits (see Feldmann *et al.*, 1992).

This algorithm is reported to suffer from excessive CPU times and large memory requirements that make simulations of huge problems slow on a single workstation or even impossible on a vector computer, because of insufficient memory capacity.

In this paper we present a parallel algorithm which performs the computation of Harmonic Balance in a distributed way on a cluster of workstations and thus reduces the inherent disadvantages of Harmonic Balance.

In Section 2, we give a short introduction to the problem and the structure of the systems of equations that are generated. Section 3 gives a general outline of how to distribute the Harmonic Balance as a whole. In Section 4, we concentrate on the most important aspect, i.e. the parallelization of the LU-decomposition. The distribution of the Jacobian is discussed in Section 5. Finally speedup results of a real life example are analysed in Section 6.

2. THE EQUATIONS OF HARMONIC BALANCE

In the following a rough derivation of the equations of Harmonic Balance is given, for more details see Feldmann (1992) or Schneider *et al.* (1993).

The network equations are generated by Kirchhoff's laws and form a system of N non-linear Differential Algebraic Equations (DAEs):

$$f(x(t), \dot{x}(t), t) := g(x(t), t) + \frac{\mathrm{d}}{\mathrm{d}t} q(x(t)) = 0,$$

with $g: \mathbb{R}^N \times \mathbb{R} \to \mathbb{R}^N$ and $q: \mathbb{R}^N \to \mathbb{R}^N$. The node potentials that are to be computed are expressed by $x \in \mathbb{R}^N$. The static part of the network equations is described by g, the dynamic part by $\frac{\mathrm{d}}{\mathrm{d}t} q(x(t))$.

For the periodic solution of this system of equations, the Harmonic Balance uses the following approach:

$$x(t) = \frac{a_0}{2} + \sum_{j=1}^{K} (a_j \cos(\omega j t) + b_j \sin(\omega j t))$$

$$\text{with } a_0, a_1, b_1, \ldots, a_K, b_K \in \mathbb{R}^N.$$

The $(2K+1)N$ variables of the $2K+1$ vectors $a_0, a_1, b_1, \ldots, a_K \in \mathbb{R}^N$ are determined by a Galerkin approach, and with the notation $X := (a_0, a_1, b_1, \ldots, a_K, b_K)^T$, the resulting equation is

$$F(X) = 0,$$

which is solved by Newton's method, starting with the value X^0. In each Newton step ν, a linear system of equations is created and solved by a block LU-decomposition followed by a forward/backward substitution. Here, each Jacobian has the structure of a sparse block matrix $(M_{i,j})_{i,j=1,\ldots,N}$, where each $M_{i,j}$ is a $(2K+1) \times (2K+1)$ submatrix. The iteration stops when the solution is of the desired accuracy. For realistic applications, the LU-decomposition of the system matrix requires more than 97% of the overall computation time, the rest includes all other operations such as model

evaluation and forward/backward substitution. Thus it makes sense to directly parallize the solver.

Figure 1 shows a typically structured Jacobian corresponding to an oscillator circuit ($N = 30$, 125 non-zero blocks).

3. GENERAL STEPS OF THE PARALLELIZATION

In this section we give a rough description of how the parallelization of Harmonic Balance works.

1. In each process the circuit simulator TITAN is started with the same input data. First a template is set up which indicates the position of non-zero blocks in the sparse Jacobian.
2. In a second step a heuristic distribution of rows to the processes is determined. The distribution is chosen such that there is an even distribution in memory and work. More details are given in Section 5.
3. Now, on each process the total model is evaluated but only those rows of the Jacobian are generated which belong to this process. Thus the immense amount of data which is responsible for more than 95% of the total memory requirement are distributed among the processes.
4. The fourth step performs the LU-decomposition of the Jacobian and the forward/backward substitution of the solution vector. For this, each process waits for the row-wise results sent to it from other processes, uses these results for its own computations, and then, in turn, distributes its own rows to the other processes. The LU-decomposition requires the largest part of the total computation time. During this step, there is a large load on the net, which is a potential bottleneck. In order to keep idle times low, the parallelization must be implemented in a rather careful way that will be presented in the next section.
5. Those parts of the solution vector belonging to the rows of the Jacobian on one process are now sent to all other processors in order to allow a new model evaluation. These communications can be performed by broadcasting.

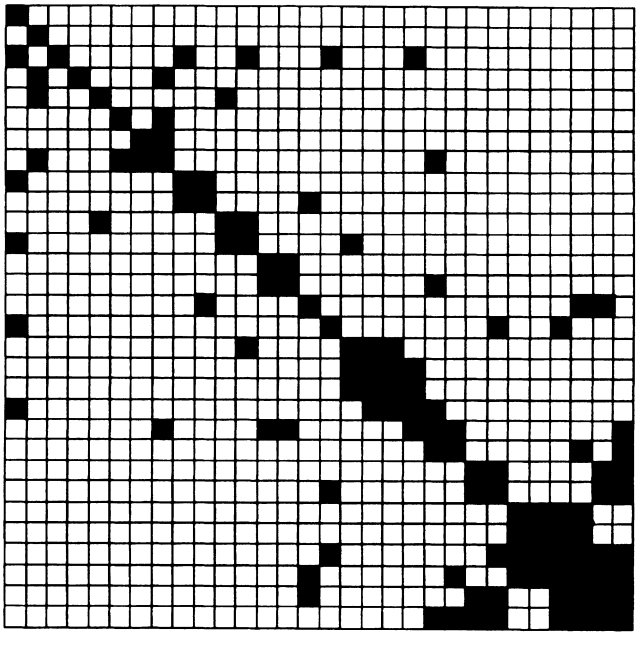

FIGURE 1. Sparse block structure of a Jacobian.

Steps 1 and 2 are initialization steps. They can be performed without any net communication. Steps 3, 4 and 5 represent the Newton iteration and have to be performed until convergence.

4. PARALLEL LU-DECOMPOSITION

In this section, all for-loops are executed as index sets (ordering insensitive) or as a list of numbers (ordering sensitive).

For the following algorithm we define:

- P is the number of processes.
- $R_i := \{M_{ik} : M_{ik} \neq 0, k = 1, \ldots, N\}$, $i = 1, \ldots, N$.
- $I_R := \{1, 2, \ldots, P\}$.
- $I_R^1 \cup \ldots \cup I_R^P = I_R$ a partition of I_R.
- $T^p := \{T_{ij}^p\}_{i,j=1,2,\ldots,N}$ with $T_{ij}^p = 1$ for M_{ij} non-zero block and $T_{ij}^p = 0$ else, $p = 1, 2, \ldots, P$.

The classical LU-decomposition is given below:

Algorithm 1:

```
for i = 1, ..., N
⌈ Q · R = M_ii
  for (j ∈ {i + 1, ..., N})
    if (M_ji ≠ 0)
      ⌈M_ji := M_ji · M_ii^-1 = M_ji · R^-1 · Q^-1
        for (k ∈ {i + 1, ..., N}):
        ⌊  if (M_ik ≠ 0)   M_jk := M_jk − M_ji · M_ik
⌊
```

For the computation of M_{ji} the QR-decomposition of M_{ii} is used. A simple way of parallelizing the LU-decomposition is shown in Algorithm 2.

Algorithm 2:

```
for i = 1, ..., N
⌈ for (p ∈ {1, ..., P}):
    ⌈ if (i ∈ I_R^p)
      ⌈ Q · R = M_ii
      ⌊ send row R_i to all other processes
    else
      wait for row R_i                                  (1)
    for (j ∈ I_R^p, j > i)                              (2)
      if (M_ji ≠ 0)                                     (3)
        ⌈ M_ji := M_ji · M_ii^-1 = M_ji · R^-1 · Q^-1
          for (k ∈ {i + 1, ..., N}):
          ⌊  if (M_ik ≠ 0)   M_jk := M_jk − M_ji · M_ik
    ⌊
⌊
```

In this algorithm, all processes are synchronized at (1), and then the j-loop is parallelized at (2). This procedure is optimal with respect to processor utilisation for a full

matrix. In a sparse matrix, after receiving row R_i, a process due to (3) may very often stay idle until receiving row R_{i+1}. This can take some time, because the process which processes R_{i+1} is often busy with the subtraction of R_i. In order to avoid these idle times, we claim that:

- Any concurrencies that arise from the sparseness of the block matrix must be detected and exploited dynamically at runtime.
- Every row where the computations are finished must be sent immediately only to those processes that need these data.
- Every computed or received result has to be processed as soon as possible. If there is any choice, working steps referring to rows with smaller numbers should be performed first.

In Algorithm 3 the meaning of T_{ij}^p is slightly changed. This change indicates the progress of computations on these rows:

$T_{ij}^p = 0$: row R_j does not need to be subtracted from row R_i.
$T_{ij}^p = -1$: row R_j must still be subtracted from row R_i.
$T_{ij}^p = 1$: row R_j has already been subtracted from row R_i.

The LU-decomposition is performed by:

Algorithm 3:
for $(p \in \{1, ..., P\})$:
⌈ *Initialize_For_LU-decomposition*
⌊ *Compute_LU-decomposition*

In *Initialize_For_LU-decomposition*, process p contains all rows R_i, $i \in I_R^p$. The index set I_d^p which contains the indices of those rows of process p having finished the computations is created as the empty set.

Initialize_For_LU-decomposition:
for $(i \in I_R^p)$
⌈ *for* $(j \in \{1, ..., i-1\})$
if $(T_{ij}^p = 1)$ $T_{ij}^p := -1$
⌊ $I_d^p := \varnothing$

Then, in *Compute_LU-decomposition*, process p repeats the following loop as long as not all rows of process p have been completely processed. (Label is referenced in *Phase_Of_Subtracting*.)

Compute_LU-decomposition:
Label: while $(I_R^p \setminus I_d^p \neq \varnothing)$
⌈ *Phase_Of_Sending*
⌊ *Phase_Of_Subtracting*

During the *Phase_Of_Sending*, at first those rows of process p from which a row must no longer be subtracted any more, but which still do not belong to I_d^p, are united in the temporary index set I_h^p. In each of these rows, the diagonal block is QR-decomposed first. Then, for this row, the set I_i^p of just those processes that can make use of row R_i is built. In this case, the corresponding process q needs row R_i, and if $q \neq p$, q is included in I_i^p. The diagonal block M_{ii} and all blocks to the right of it in row R_i are sent to all processes in I_i^p. Finally, all rows of I_h^p are included in I_d^p, because the computations are completed.

Phase_Of_Sending:
⌈ $I_h^p := \{i \in I_R^p \setminus I_d^p : T_{ij}^p \neq -1, \forall j < i\}$
for $(i \in I_h^p)$
⌈ $M_{ii} = Q \cdot R$
$I_i^p := \{q \in \{1, ..., P\} \setminus \{p\} : \exists j \in I_R^q : T_{ji}^p = -1\}$
⌊ *for* $(q \in I_i^p)$ *send row* R_i *to processor* q
⌊ $I_d^p := I_d^p \cup I_h^p$

In the *Phase_Of_Subtracting*, the set I_v^p is built containing all indices of rows which are completely computed and which process p can handle. I_v^p is the union of I_d^p and I_s^p which must be evaluated at this time. In I_s^p all rows are registered that are sent to p from the other processes and that have not yet been removed by process p. Then, for each element i of I_v^p in ascending order the set I_{su}^p (indices of rows) is built up that belong to p and from which row R_i can now be subtracted. Then, where R_i is external, i.e., was sent from another process, a check is made to determine whether there remains a row of process p to which R_i can be applied. If not R_i is removed. Then the *Phase_Of_Subtracting* is quit by a jump to *Label* at the start of *Compute_LU-decomposition.*

Phase_Of_Subtracting
⌈ *Determine* I_s^p
$I_v^p := I_d^p \cup I_s^p$
for $(i \in I_v^p)$
⌈ $I_{su}^p := \{j \in I_R^p \setminus I_d^p : T_{ji}^p = -1$
$\wedge (T_{jk}^p \neq -1 \vee T_{ki}^p = 0, \forall k < i)\}$
if $(I_{su}^p \neq \varnothing)$
⌈ *for* $(j \in I_{su}^p)$
⌈ $M_{ji} := M_{ji} \cdot R^{-1} \cdot Q^{-1}$
for (k ∈ {i + 1, ..., N})
if $(T_{ik} \neq 0)$ $M_{jk} := M_{jk} - M_{ji} \cdot M_{ik}$
⌊ $T_{ji}^p := 1$
if $(i \notin I_R^p)$
⌈ $I_{h1}^p := \{j \in I_R^p : j > 1, T_{ji}^p = -1\}$
⌊ *if* $(I_{h1}^p = \varnothing)$ *remove row* R_i
⌊
⌊
⌊ *goto Label*

The following properties together with the uniqueness of the LU-decomposition imply the correctness of the algorithm:

1. The algorithm terminates after $(N+1)N/2$ steps because after each step the number of $\{\cup_{p=1}^P I_q^p\}$ or $\{\Sigma_{p=1}^P \Sigma_{i,j=1}^N T_{ij}^p\}$ increases.
2. Each update operation is an equivalence transformation in the sense of linear algebra.
3. After termination, all blocks under the diagonal one are updated $(T_{ij}^p \neq -1, \forall p = 1, ..., P, i, j = 1, ..., N)$ and all diagonal blocks are QR-decomposed $(I_d^p = I_R^p, \forall p = 1, ..., P)$.

5. THE DISTRIBUTION OF THE ROWS

The efficiency of the outlined algorithm depends strongly on the distribution of rows among the processes. Efficient exploitation of memory demands an even distribution of blocks among all processes. Even distribution of work requires a more sophisticated strategy. For instance, we have to prevent that successive rows are computed by the same process. Otherwise, according to algorithm 3 from Section 4, after a row R_i is subtracted from a row R_j, R_j would not be sent despite being ready for it, because the process that possesses R_j is busy with subtracting R_i from R_{j+1}, R_{j+2} and so on. This measure also increases the degree of concurrency of elimination steps on different rows for each process.

These requirements are realized by the following heuristic algorithm.

1. Define the average number of non-zero blocks in a row and of blocks on a process:
   A_R:=(*number of full entrances of the block matrix*)/N
   A_P:=(*number of full entrances of the block matrix*)/P
2. Distribute the rows with more than $C \times A_R$ non-zero entries. (C is an experimentally determined value: $C = 5$ for the example in Section 6)
3. Distribute the remaining rows with a high or a middle index number:
   $i := N$
   repeat
   ⌈ B:={*remaining processes with less than* A_P *blocks*}
   for ($j \in \{1, \ldots, |B|\}$):
   contribute row R_{i-j+1} *to process* j
   ⌊ $i := i - |B|$
   until($i < N/3$)
   Change the process number of successive rows until no process has more than $A_P + 2$ *blocks.*
4. Distribute the remaining rows such that every process has between $A_P - 2$ and $A_P + 2$ submatrices.

Other strategies may try to achieve an even distribution of the number of matrix operations or the number of external blocks that every process must read. By transforming the block matrix according to nested dissection as shown in George (1973), we can also attempt to reduce the network load and interprocess data dependences. But for real life simulations, the computation on single rows becomes dominant, if many computers are employed. Then, for any strategy, further splitting of blocks or rows is necessary.

6. RESULTS AND DISCUSSION

6.1. Hardware

Our algorithm has been implemented on a workstation cluster at the Institut für Informatik of the Technische Universität München. It consists of Hewlett-Packard HP720 workstations that are connected by an Ethernet at a transfer rate of 10 MBit/s. At the chair of Professor Zenger, in a parallelization of another algorithm which has far less communication and inherent data dependence, 110 workstations achieved a performance of more than 1 GFlops (see Griebel *et al.*, 1993).

For our implementation we used a cluster of 16 workstations. Each HP has about 16 MFlops and a free main storage of about 15 MByte (see Bonk and Rüde, 1992). On 14 HPs, there is an additional swap space of 15 MByte, 2 HPs have an extended swap space of 300 MByte. We achieved 8 MFlops for matrix multiplications, and less for the QR-decompositions and for the calculation of $M_{ji} \cdot M_{ii}^{-1}$ on a single machine.

6.2. Network communication

Almost all net communication takes place during the LU-decomposition. After the computations in a matrix row are finished, those blocks to the right of the diagonal block are sent to the process utilizing these data. In our real life examples, there are only two or three such processes, because of the sparseness of the matrix. In this test example with a matrix size of 200 MByte, 250 MByte have to be sent. Due to the occurrence of collisions, it is realistic to assume that on an Ethernet the transmission of 250 MByte takes between 500 and 1000 s. Even if the distributed LU-decomposition requires more than 1000 s, the fact that the row transmission is considerably delayed will increase the idle times of the processes and, consequently, the total computation time significantly.

In order to reduce the load of the network at the Institut für Informatik at the Technische Universität München we started to implement a broadcast procedure based on the UDP protocol. It allows each data package to be transferred only once, and in the course of this one circulation, it is read by all interested processes. Thus, in our test example only 100 MByte would have to be transmitted. First experiences with our broadcast procedure make us expect that the data will then be transferred on the Ethernet within about 150 s.

Since this broadcast procedure is not fully available, we decided to send the floating point numbers with only 4-byte accuracy. This decision is based on the assumption that the corresponding rounding errors caused thereby can be balanced out by a few extra Newton iterations. In our example, the transmission of the resulting 125 MByte of data can be handled by the Ethernet even with collisions in 250 to 500 s.

6.3. Test results

The efficiency of the parallel implementation has been demonstrated on an analog converter circuit. It consists of 86 bipolar transistors and has 247 nodes, i.e. $N = 247$. 1631 blocks of the Jacobian are non-zero. During the LU-decomposition 247 QR-decompositions, 696 matrix divisions and 2160 matrix multiplications must be performed. In most tests, the number of frequencies K is set to 59, that is the dimension of one block is 119. Then, each block has a size of about 120 KByte if the

TABLE 1. An LU-decomposition with different K on 15 workstations

Number of frequencies K	19	29	39	49	59	69	79
Computation time (s)	84	180	264	396	550	740	960
Storage needed (MByte)	22	49	88	113	198	265	352
Floating point operations (GFlop)	0.35	1.25	2.31	6.01	10.5	16.80	25.21

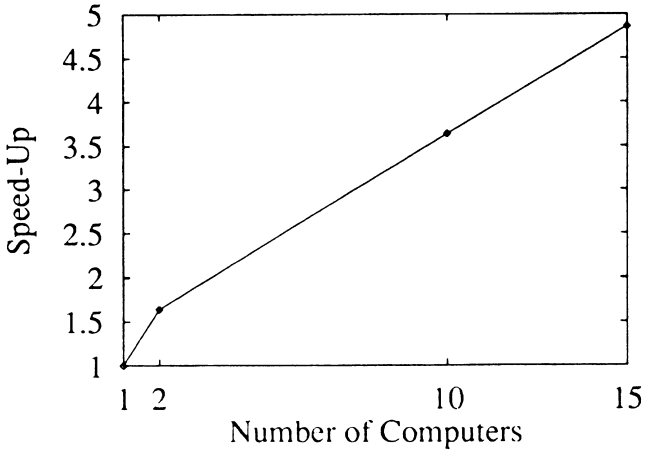

FIGURE 2. Speed-up results for a entire Newton iteration.

floating point numbers are given in 8-byte accuracy. A QR-decomposition of one block needs 1.1 MFlop, each of the others 3.4 MFlop. Due to the different code optimizations, a matrix multiplication of these blocks needs 0.4 s, and a QR-decomposition and a matrix division 1.0 s. The total amount of data of the system matrix is 198 MByte and 10.5 GFlop must be executed for a LU-decomposition.

We will present the speedup results of the computation of an entire Newton iteration including the LU-decomposition for our test problem with $K = 59$ on 1, 2, 10 and 15 workstations. The measured computation times are 102, 62, 28 and 21 min, respectively. The corresponding speed-up diagram is given in Figure 2.

The fact that the speedup results do not scale with the number of processors is due to the chosen example. Many circuit elements accessing the same node produce a row in the Jacobian with very many non-zero blocks. Here one row has 84 non-zero blocks in contrast to six blocks on average. Therefore no even distribution of work can be achieved for several processors.

For larger examples ($N > 247$) this disadvantage can be balanced out by a reasonable distribution of rows.

The various K values and the corresponding times are given in Table 1 that also includes the corresponding memory and CPU requirements.

7. CONCLUSION

The Harmonic Balance is one of the most general methods of non-linear frequency analysis. The inherent disadvantages of this method—enormous memory and CPU requirements—can be compensated by parallel implementation on a cluster of workstations. Since the Jacobian is created, stored and worked on in a distributed way, it is possible to solve problems that need extensive memory capacity. Since the sparseness of the block matrix is additionally exploited during the LU-decomposition good speedup results have been achieved.

Acknowledgements

The authors would like to thank Professor Zenger and the department of Dr Gilg for many helpful discussions.

REFERENCES

Bonk, T. and Rüde, U. (1992) *Performance Analysis and Optimization of Numerically Intensive Programs.* SFB-Bericht 342/26/92 A, TU München, Germany.

Feldmann, U., Wever, U., Zheng, Q., Schultz, R. and Wriedt, H. (1992) Algorithms for modern circuit simulation. *Archiv für Elektronik und Übertragungstechnik*, **46**, 274–285.

George, A. (1973) Nested dissection of a regular finite element mesh. *SIAM J. Numer. Anal.*, **10**, 345–363.

Griebel, M., Huber, W., Störtkuhl, T. and Zenger, C. (1993) On the parallel solution of 3D PDEs on a network of workstations and on vector computers. *Springer Lecture Notes*, in press.

Schneider, M., Wever, U. and Zheng, Q. (1993) Parallel Harmonic Balance. Paper presented at *VSLI-Conf. 1993*, Grenoble, France.