Parallel sparse-matrix solution for direct circuit simulation on a transputer array

A. Mahmood
Y. Chu
T. Sobh

Indexing terms: VLSI circuits, Sparse-matrix solutions, Transputers, Simulation

Abstract: Sparse-matrix solution is a dominant part of execution time in simulating VLSI circuits by a detailed simulation program such as SPICE. The paper develops a parallel-block partitionable sparse-matrix-solution algorithm which exploits sparsity at the matrix block level as well as within a nonzero block. An efficient mapping scheme to assign different matrix blocks to processors is developed which maximises concurrency and minimises communication between processors. Associated reordering and efficient sparse storage schemes are also developed. Implementation of this parallel algorithm is carried out on a transputer processor array which plugs into a PC bus. The sparse matrix solver is tested on matrices generated from a transistor-level expansion of ISCAS-85 benchmark logic circuits. Good acceleration is obtained for all benchmark matrices up to the number of transputers available.

1 Introduction

The design of modern high-speed analogue and digital VLSI systems involves extensive circuit-level simulations. A circuit simulator allows a design to be tested thoroughly and its timing behaviour analysed in detail before committing it to silicon. SPICE [1] is one such general-purpose circuit simulation program which is widely used. It is capable of doing nonlinear DC nonlinear transient and linear AC analyses. Note that even though a circuit design may be completely digital, circuit simulations are important in analysing critical delays in the system. Further, as the clock rate of digital designs is increasing, the digital circuitry is behaving more like an analogue circuit whose operation can only be tested by a detailed circuit-simulation program such as SPICE.

Because a circuit-simulation program provides very good detail in simulation of circuits, it consumes high amounts of CPU time in simulating large circuits [$O(n^3)$ for a circuit with n nodes]. Hence simulations become too time consuming for circuits with more than a few thousand transistors. The transistor count in newer VLSI processors is already in the millions and rapidly increasing every year. The execution time of circuit simulation can be improved by exploiting the parallelism in the CPU time-intensive parts of the simulation program.

Circuit simulation involves two numerically intensive steps. The first step, known as the LOAD phase, evaluates the circuit models and assembles the matrix after a nodal analysis of the circuit. The second step, known as the SOLVE phase, solves the resulting sparse-matrix equation to obtain the unknown nodal voltages. These two steps are repeated many times during a transient analysis as the simulation time advances in small increments. The LOAD phase has been found to be relatively easily 'parallelisable'. On the other hand, the SOLVE phase involves a sparse-matrix solution whose highly sparse, unsymmetric and unstructured nature makes parallelisation very difficult [2–4]. This paper focuses only on the parallelisation of the SOLVE phase.

There are two approaches followed in circuit simulation known as the 'direct' and 'relaxation' methods. Although the relaxation method, which is based on the Gauss–Seidel iterative-matrix-solution technique, often converges faster to a solution compared with the direct approach (and also has better parallelism), it has problems in simulating bipolar and close feedback MOS circuits [5]. For this reason, more popular simulators such as SPICE, ASTAP, SABER, MISIM etc. use the direct method which provides accurate simulation for all different kinds of circuits.

Quite a few attempts have been made at parallelising circuit simulation in the last decade [2–11] varying from a fine-grained approach on systolic arrays [10] to coarse-grained approaches on a network of workstations [11]. Of these, most success has been reported in a theoretical study on systolic arrays [10], while the coarse-grained approach implemented on a network of workstations using PVM did not produce any acceleration [11]. Implementations on general-purpose parallel machines have reported some success [2–4, 6–8]. However, the high cost of general purpose parallel computers is causing them to fall out of favour in recent years. Because of cost considerations and because of the fact that a fine-grain approach to parallel-circuit simulation has indicated the highest potential for acceleration, an

IEE Proceedings online no. 19971566

Paper first received 15th November 1996 and in revised form 4th August 1997

A. Mahmood and T. Sobh are with Computer Science and Engineering, University of Bridgeport, Bridgeport, CT 06601, USA

Y. Chu is with Washington State University at Tri-Cities, Richland, WA 99352, USA

implementation of circuit simulation on a low-cost array of transputer processors was pursued.

2 Parallel SOLVE for direct circuit simulation

There are four basic steps in a circuit-simulation program such as SPICE [1]. The first step is the nodal-equation formulation for a given circuit. If there are nonlinear elements in the circuit, the node equations result in a system of ordinary differential equations. These differential equations are then converted to a system of nonlinear equations by approximating the derivatives with backward differencing. In the third step, the Newton–Raphson technique is applied to convert the nonlinear equations to a linear form. The matrix equation generated from this is then solved repeatedly (in the SOLVE phase) for the unknown nodal voltages for each time step in the simulation. The zero–nonzero structure of the sparse matrix for a given circuit is entirely dependent on how the circuit elements are interconnected.

The SOLVE phase has a sparse matrix equation of the form $\mathbf{A}\,\mathbf{x} = \mathbf{b}$, where $\mathbf{A}$ is an $n \times n$ matrix and $\mathbf{x}$ and $\mathbf{b}$ are $n \times 1$ vectors. Although Gaussian elimination can be used to solve this matrix equation, it is not efficient in cases where the coefficient matrix $\mathbf{A}$ stays the same but the $\mathbf{b}$ vector changes many times. A variation of the Gaussian elimination is LU decomposition [also time complexity $O(n^3)$ for an $n \times n$ matrix] in which the $\mathbf{A}$ matrix is transformed into a product of lower and upper triangular matrices. After $\mathbf{A}$ has been decomposed into its $\mathbf{L}$ and $\mathbf{U}$ factors, two triangular equations $\mathbf{L}\mathbf{y} = \mathbf{b}$ and $\mathbf{U}\mathbf{x} = \mathbf{y}$ are solved to find the unknown xs. The solution of a triangular equation has a time complexity of $O(n^2)$.

To achieve good acceleration in the parallelisation of the SOLVE phase, it is necessary to partition the matrix onto different processors such that it results in concurrent execution with good load balance as well as reduced communication between processors. In the past, the partitioning has been carried out indirectly using the node-tearing approach [12], and has resulted in relatively low accelerations due to the load-balancing problem [2]. This occurs when the subcircuits are not of identical size, causing waiting on completion of the solution of the largest subcircuit. In this work, a direct partitioning of the matrix into equal size blocks for mapping on to available processors is followed. Optimal partitioned algorithms for solving a dense linear system of equations have been developed previously [13, 14]. However, the unstructured and highly sparse matrices encountered in circuit simulation make this a difficult problem. A sparse-parallel-dense-matrix-LU-decomposition algorithm will be developed based on a modification of the parallel-dense-matrix scheme of [13].

In the parallel-LU-decomposition scheme of [13], a given matrix $\mathbf{A} = (a_{ij})$ is partitioned into k^2 submatrices each of size $m \times m$ (each submatrix is referred to as a block). The parallel algorithm first computes the LU decomposition in block $A_{0,0}$ using a serial algorithm. Then after computation of $L_{0,0}^{-1}$ and $U_{0,0}^{-1}$, in step 2, all remaining blocks in the first row and first column can be computed in parallel for their U and L computations, respectively. Similarly step 3 computes the LU decomposition in the diagonal block $A_{1,1}$ followed by computation of $L_{1,1}^{-1}$, $U_{1,1}^{-1}$. Step 4 concurrently computes the U factors in row 1, and L factors in column 1. The computation proceeds in a similar manner until the LU decomposition is performed in the last diagonal block in step $2k-1$. The pseudocode for the dense-matrix-partitioned-LU-decomposition algorithm is as follows:

Parallel partitioned-dense-matrix-LU-decomposition algorithm

```
PARALLEL_PARTITIONED_DENSE_LUD(A)
  for q = 0 to K–1      /* K = number of blocks in a row/column */
    if (q > 0) then compute_A_prime(A,q,q)
    Decompose A_{q,q} into L_{q,q}.U_{q,q}
    if (q < (K–1)) then
          Compute inverse matrices L_{q,q}^{-1} and U_{q,q}^{-1}
          /* S1 and S2 are computed in parallel */
          S1:  for all p = (q+1) to K–1 do
                 if (q > 0) then compute_A_prime(A,p,q)
                 compute L_{p,q} = A_{p,q} . U_{q,q}^{-1}
               endfor
          S2:  for all p = (q+1) to K–1 do
                 if (q > 0) then compute_A_prime(A,q,p)
                 compute U_{q,p} = L_{q,q}^{-1} . A_{q,p}
               endfor
  endfor
end PARALLEL_PARTITIONED_DENSE_LUD

compute_A_prime(A,p,q)
  A' = 0; r = min(p,q)
  for s = 0 to r–1 do
    A' = A' + L_{p,s} . U_{s,q}
  A_{p,q} = A_{p,q} – A'
end compute_A_prime
```

As an example, $L_{4,3}$ is computed by first determining the A-prime $(A_{4,3})'$ as, $(A_{4,3})' = A_{4,3} - (L_{4,0}\,U_{0,3} + L_{4,1}\,U_{1,3} + L_{4,2}\,U_{2,3})$. Then, $L_{4,3} = (A_{4,3})'(U_{3,3})^{-1}$.

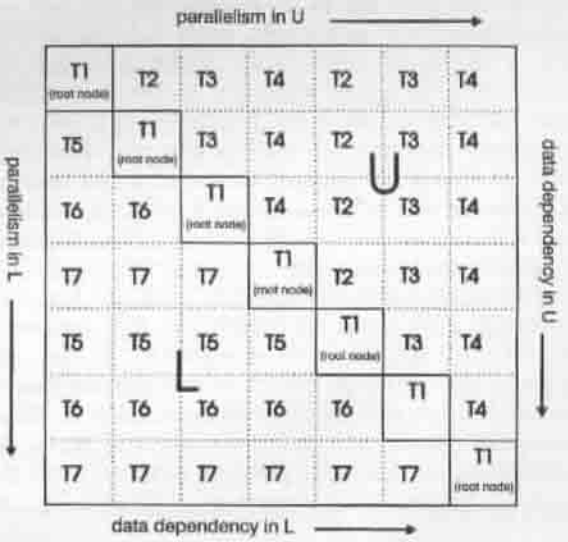

Fig. 1 *Execution sequence for block-partitioned parallel LU decomposition*

The number of floating point arithmetic operations required by the above parallel-LU-decomposition algorithm is very close to that of the conventional serial algorithm. The count for the parallel algorithm decreases slightly as the block size is made smaller. Overall, it is within 10% of the count required in the conventional serial algorithm. For example, the number of exact normalised floating-point operations in the LU decomposition of a 500×500 matrix is 125 124 250 for the conventional serial algorithm, and 132 646 330 for the parallel algorithm with a block size of 20×20. The normalised count assumes a weight of 2 for floating-point multiplication and 4 for division.

2.1 Partitioned-sparse-matrix LU decomposition

The partitioning method to perform LU decomposition on a transputer array is illustrated in Fig. 1. The partitioned matrix-block-to-processor assignment has been chosen such as to maximise concurrency and minimise communication between processors. The given matrix $\mathbf{A} = (\mathbf{a}_{i,j})$ is partitioned into equal size blocks of order $m \times m$.

After a diagonal block $A_{i,i}$ has been decomposed into its $L_{i,i}$ and $U_{i,i}$ factors by the root transputer node (labelled T1 in Fig. 1), it communicates the $(L_{i,i})^{-1}$ to all the processors assigned to blocks in the ith row, and $(U_{i,i})^{-1}$ to the processors assigned to the ith column blocks. These processors then perform the L and U computations concurrently, as indicated by the parallel algorithm of Section 2.

Owing to the low connectivity of each node in a large circuit, the matrices generated in circuit simulation tend to be highly sparse (> 99% zeros). Thus the dense parallel algorithm needs to be modified to take advantage of the sparsity in the problem. A sparse-matrix solution has been developed in [10] for use on systolic arrays which exploits the sparsity at the block level only, i.e. it skips computations related to zero blocks. It was shown in [10] that, if the block size is kept small (2×2), excellent accelerations can be obtained by using multiple 2×2 systolic array modules. However, the results deteriorate rapidly as the block size increases above 2×2 because of decrease in block-level sparsity.

In this paper, a two-level sparse algorithm is developed which exploits sparsity at the block level as well as within a block. Note that a block size of 1×1 results in maximum concurrency but also maximum communication between processors. At the other extreme, if block size is $n \times n$, then the problem is reduced to a sequential execution. As the block size increases gradually above 2×2, the concurrency and the communication between processors decrease, the block-level sparsity decreases, but the sparsity within each block increases. Hence an optimum block size is a function of the sparsity in the problem, speed of each processor, and the communication between processors. Thus systolic arrays in which processors provide data transfer of one word in each clock cycle on all their links would run a matrix solver efficiently with a small block size so as to maximise concurrency. However, slower communication environments with high latency such as a network of workstations connected by Ethernet would require a much larger block size. Transputer arrays fall in between the above two extremes, requiring a medium granularity in LU decomposition for its efficient parallel execution.

To exploit the sparsity at the block level, most of the fill-in (zero blocks which get changed) can be decided according to the position of a zero block in the original matrix. The sparse algorithm developed here uses a 2-D one-bit array called Az to store the information about zero–nonzero blocks. If Az is checked, L and U computations for blocks which do not cause fill-in, can be skipped. If a fill-in occurs at $\text{block}_{i,j}$, but $A_{i,j}$ is originally zero, then $Az_{i,j}$ is set to 1 during the computations for this block.

To exploit the sparsity within a nonzero block, sparse routines can be developed to save on computations returning zero results. The implementation of the sparse routines requires a fast access to the next nonzero elements. This is achieved by using a compressed-sparse-storage scheme {compressed sparse-row form (CSR) and compressed sparse-column form (CSC) [15, 16, 19]} within the blocks. The compressed-sparse-row (CSR) format uses three arrays to store an $n \times n$ sparse matrix with q nonzero entries:

(i) A $q \times 1$ **X** array contains the nonzero elements in each row.

(ii) $q \times 1$ array **XJ** that stores the column numbers of each nonzero element.

(iii) A $n \times 1$ array **XI**, the ith entry in **XI** points to the first entry of the ith row in **X** and **XJ**.

$$\begin{bmatrix} 1 & 0 & 0 & 2 & 0 & 3 \\ 4 & 5 & 0 & 0 & 0 & 0 \\ 0 & 6 & 7 & 0 & 0 & 8 \\ 9 & 0 & 0 & 10 & 11 & 12 \\ 0 & 13 & 0 & 0 & 14 & 0 \\ 0 & 0 & 0 & 0 & 0 & 15 \end{bmatrix}$$

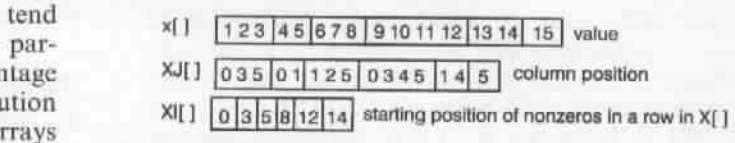

Fig. 2 *CSR storage of a 6×6 sparse matrix*

Fig. 2 shows an example for the CSR storage scheme. The compressed-sparse-column (CSC) form is related to CSR form such that the roles of rows and columns are reversed.

Since the time-intensive computations in the parallel-LU-decomposition algorithm are block-matrix multiplication, and LU decomposition in the diagonal blocks, sparse routines for these operations are implemented [sparse_matrix_multiply( ) and sparse_lu_gauss( )]. The sparse-matrix-multiply( ) routine stores the first matrix block to be multiplied in CSR form so that the next nonzero in a row can quickly be accessed. Similarly, the second block to be multiplied is stored in CSC form which provides a quick access to the nonzeros in each column of the matrix. The sparse_lu_gauss( ) routine speeds up the serial LU decomposition needed in the diagonal block by skipping the zero subtractions in row updates. This is achieved by storing the upper triangular part of the diagonal block in CSR form to help skip computations due to a zero operand. The block is also kept in dense-storage form so that a quick access to the actual nonzero entries can be made. The pseudocode for the sparse_lu_gauss( ) routine is as follows:

```
void sparse_lu_gauss(B,X,XI,XJ)   // B[][] is the diagonal block
{                                 // X, XI, XJ contain the CSR storage form for B[][]
```

```
for (i = 1; i < BLOCK_SIZE; i++)
 {
   for (j = 0; j < i; j++)
     if (B[i][j] != 0)
       rowupdate_CSR(M,XI,XJ,i,j); // sparse row
       update in row i to
 }                          // zero out position B[i][j]
                            (Gaussian elimination)
void rowupdate_CSR(B,XI,XJ,i,j)  // fast row update
                                 to zero out B(i,j) by
{ // exploiting sparsity. B has a copy of it stored in
CSR form for this purpose
  B[i][j] = B[i][j]/B[j][j] // multiplier in row update
  first_nonzero_pos = XI[j]; last_nonzero_pos =
  XI[j+1] // # nonzeros in row j
  for (k = first_nonzero_pos +1; k < last_nonzero_
  pos; k++)
    B[i][XJ[k]] = B[i][XJ[k]] – B[i][j] * B[j][XJ[k]]
}
```

Note that the conversions needed from dense-storage form to CSR or CSC form for an $m \times m$ block are $O(m^2)$, whereas dense-matrix multiplication or LU decomposition require $O(m^3)$ operations. Thus the conversion overhead is minor compared with the execution-time savings from exploiting sparsity. Further, only one conversion is required for operating on an entire row or column of blocks, e.g. $(L_{i,j})^{-1}$ is converted only once to CSR form and transmitted to all the corresponding row blocks for their U computations. The two-level-parallel-sparse-LU decomposition is as follows:

Two level-sparse-parallel-LU-decomposition algorithm

```
read_input_matrix( )   // Read and store matrix in
                       sparse form
                       // Store block level sparsity
                       info in 2-D array Az
For i = 0 to K–1       // K is the total number of
                       blocks in a row/col.
  For j = 0 to K–1
    if (i = = j)       // Compute L and U factors for
                       diagonal blocks
      AU_to_CSR(A_i,j, X,XI,XJ) // change upper
                                   triang. to CSR
      if (i > 0 )      // not the first
                       row or column
        compute_A_prime(A,Az,i,i,i)
      sparse_lu_gauss(A_i,j, X,XI,XJ) // sparse LU
                                         decomp. in A_i,i
      if (i < K–1) // Not the last diagonal block
        L_invert(A_i,j, L^-1_i,i) // compute
                                     L^-1_i,i
        U_invert(A_i,j, U^-1_i,i) // compute
                                     U^-1_i,i
        L_to_CSR(L^-1_i,i,X,XI,XJ)
                                  // change to CSR
        U_to_CSC(U^-1_i,i,Y,YI,YJ)
                                  // change to CSC
    if (j > i)         // U computations in upper tri-
                       angular blocks
      if ((i > 0 ) || (j > 0)) // not the first row or
                                  column
        compute_A_prime(A,Az,i,i,j)
      if (Az_i,j = 1) // A_i,j is nonzero block
        dense_to_CSC(A_i,j,C,CI,CJ)
        sparse_matrix_multiply(A_i,j,
        X,XI,XJ,C,CI,CJ)
    else               // L computations in lower tri-
                       angular blocks
      if ((i > 0 ) || (j > 0)) // not the first row or
                                  column
        compute_A_prime(A,Az,j,i,i)
      if (Az_j,i = 1) // A_j,i is nonzero block
        dense_to_CSR(A_j,i,R,RI,RJ)
        sparse_matrix_multiply(A_j,i,
        R,RI,RJ,Y,YI,YJ)
```

The source code and detailed explanation for the different routines used in the above program can be found in [16]. An example is presented in Fig. 3 to clarify the steps involved.

$$A = \begin{bmatrix} X_{0,0} & O_{0,1} & O_{0,2} & X_{0,3} \\ O_{1,0} & X_{1,1} & O_{1,2} & O_{1,3} \\ O_{2,0} & X_{2,1} & X_{2,2} & O_{2,3} \\ O_{3,0} & O_{3,1} & O_{3,2} & X_{3,3} \end{bmatrix}$$

Fig. 3 *Sparse matrix partitioned into 4 × 4 blocks*

Fig. 3 shows a sparse matrix partitioned in 4 × 4 blocks. Each block is marked as **X** or **O**. **X** means a nonzero block (i.e. it has some nonzero elements), and **O** means the entire block is zero. The execution sequence is described in the following steps.

Step 1: $L_{0,0}$ and $U_{0,0}$ factors of $A_{0,0}$ are computed by the sparse_lu_gauss( ) routine. Then inverses of $L_{0,0}$ and $U_{0,0}$ are computed. $(U_{0,0})^{-1}$ is converted to CSC form and sent to the processors assigned to the first column. This is done to exploit the sparsity in multiplication as the L computations require a multiplication by $(U_{0,0})^{-1}$. Similarly $(L_{0,0})^{-1}$ is converted to CSR form and sent to the first row processors.

Step 2: U factors of the first row blocks are computed as $U_{0,1} = (L_{0,0})^{-1} A_{0,1}$, $U_{0,2} = (L_{0,0})^{-1} A_{0,2}$, $U_{0,3} = (L_{0,0})^{-1} A_{0,3}$. Here, since $A_{0,1} = A_{0,2} = 0$ (by checking the Az array), computations for $U_{0,1}$ and $U_{0,2}$ are skipped as they are zero. L factors of the first column are computed by $L_{1,0} = A_{1,0} (U_{0,0})^{-1}$, $L_{2,0} = A_{2,0} (U_{0,0})^{-1}$, $L_{3,0} = A_{3,0} \cdot (U_{0,0})^{-1}$. Here, since $A_{1,0} = A_{2,0} = A_{3,0} = 0$, the computation for $L_{1,0}$, $L_{2,0}$, $L_{3,0}$ are skipped.

Steps 3, 4: L and U factors of $A_{1,1}$ are computed and inverses sent to appropriate processors. Then U computations in row 1 and L computations in column 1 are performed. Sparsity at the block level and within a block is exploited.

Steps 5, 6: L and U factors of $A_{2,2}$ are computed and inverses sent to appropriate processors. Then U computations in row 2 and L computations in column 2 are performed. Sparsity at the block level and within a block is exploited.

Step 7: L and U factors in the last diagonal block are computed by $A'_{3,3} = A_{3,3} - (L_{3,0} U_{0,3} + L_{3,1} U_{1,3} + L_{3,2} U_{2,3})$ (compute_A_prime routine). Since $L_{3,0} = L_{3,1} = L_{3,2} = U_{1,3} = U_{2,3} = 0$, $A'_{3,3} = A_{3,3}$. Thus, computation time is saved for $A'_{3,3}$. Sparse_lu_gauss( ) routine then exploits the sparsity within $A'_{3,3}$ and factors it into $L_{3,3}$ and $U_{3,3}$.

The processor-to-block assignment as shown in Fig. 1 can now be explained and justified easily. As indicated by the partitioned algorithm, $U_{0,2}$ is computed as $U_{0,2} = (L_{0,0})^{-1} A_{0,2}$. Similarly $U_{1,2}$ requires $U_{0,2}$ in its computation ($U_{1,2}$ is computed from $A'_{1,2}$ which is equal to $A'_{1,2} = A_{1,2} - L_{1,0} U_{0,2}$). Thus the communication can be reduced if the processor computing $U_{0,2}$ is also assigned to block $A_{1,2}$ to compute $U_{1,2}$ in it. For this reason, the blocks in a column in the upper-triangular part of the matrix are always assigned to the same processor. Similarly, the blocks in a row in the lower-triangular part of the matrix are assigned to the same processor to perform the L computations in order to minimise the communication. The different row and column blocks in a parallel step are assigned to different processors to exploit the concurrency in the problem.

The two-level sparse algorithm and the processor-to-block-assignment scheme as described in Fig. 1 further help the load-balancing problem by minimising the effect of many zero matrix blocks being assigned to one processor during a parallel step. Since the block size is small compared with the original matrix, there are many blocks being executed by a processor during a parallel step of the algorithm. Also, each processor is periodically assigned a block over the total number of blocks in a row or column of the matrix. This improves the probability of each processor being assigned a similar number of zero or nonzero blocks to the other processors. Further, by exploiting the sparsity within a nonzero block, the potential of load imbalance is reduced.

The solution has also been parallelised for two triangular equations ($\mathbf{Ly} = \mathbf{b}$, $\mathbf{Ux} = \mathbf{y}$) which are needed after the LU decomposition [16]. Owing to space constraints, only the parallel LU decomposition [which is more time consuming, $O(n^3)$] is discussed in this paper.

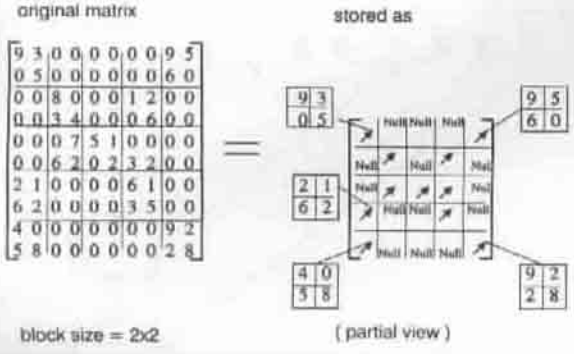

Fig. 4 *Storage of a matrix for two-level sparse execution*

2.2 Sparse-matrix storage to conserve memory

The CSR and CSC storage schemes described earlier are used primarily to speedup execution of sparse routines. However, it is also important that the overall matrix be stored efficiently so that solution of large problems is feasible. Matrices generated in the simulation of large circuits are highly sparse and may require enormous amounts of memory if stored in dense form. For example a matrix representing a circuit with 10000 nodes would require 400 Mbyte of RAM if stored as a regular 2-D array. Thus a sparse-storage scheme is required which conserves memory while still allowing efficient access to the nonzero elements. In the past, several storage schemes have been developed ranging from a simple linked-list type of storage, as implemented in the Berkeley SPICE code, to complex overlapped scattered arrays [17].

A new storage scheme has been developed for the present two-level sparse matrix solver. To conserve memory, a dynamically allocated storage for nonzero blocks is carried out. A 2-D pointer array is maintained in which a pointer points either to the nonzero block or to NULL if the block is zero. In this way, a fast access to nonzero blocks can readily be obtained. An example is shown in Fig. 4.

The above storage scheme not only conserves memory but provides very fast access to elements of a nonzero block. Since the block-level sparsity is found to be very high (>90% for block sizes < 10 × 10) for large problems, the memory-saving properties of this scheme are very good. Note that the nonzero blocks are stored in noncompact form for fast access during the block-level sparse-matrix solution. However, this noncompact form is changed to compact form (CSR/CSC) to utilise sparsity within a block during execution. The scheme described here is also recursive in nature, i.e. the 2-D pointer array can be made to point to a large block which in turn is stored in terms of another pointer array and dynamically allocated smaller nonzero blocks. Such an indirect scheme may be needed in simulation of very large circuits.

2.3 Matrix reordering to minimise fill-in

Reduction of fill-in is an important factor in maximising the gains in sparsity utilisation. A matrix reordering, i.e. relabelling the nodes, can be carried out to reduce the fill-in and thus execution time. In the development of the SPICE circuit-simulation program [1], it was concluded that the Markowitz reordering algorithm [18] provides good reordering with substantially less computational effort than other approaches. The Markowitz algorithm works as follows. Select a nonzero entry of the active submatrix and bring it to the pivoting position by means of the necessary row and column exchanges in such a way that the product $n(c)n(r)$ is kept at a minimum, where $n(c)$, $n(r)$ are, respectively, the number of nonzero entries in a column and row of the active submatrix for the entry selected. The active submatrix is the portion after the pivot point.

Since this matrix solver takes advantage of the sparsity at the block level, the element-level Markowitz algorithm as described in [18] needs to be modified to obtain a minimum fill-in at the block level. The present block-level reordering algorithm always selects the next pivot to be a diagonal block whose row–column nonzero product is the smallest. The nonzero block count for a row or column is determined from the entire matrix and not just the active submatrix, as is usually done in the Markowitz algorithm at the element level. The reason is that the Markowitz algorithm is usually used in conjunction with the solution of a matrix. However, in the present case, the matrix is reordered completely before an attempt to solve it is made. Since the zero nonzero structure of the original matrix is dependent on the physical circuit interconnections, the Markowitz reordering is a one-time overhead in the solution of a matrix. The algorithm itself is quite simple and takes much less execution time than the overall matrix-solution time.

3 Implementation on transputers

The parallel-sparse matrix-solution algorithm as described in Section 2 has been implemented on an array of transputers. The implementation hardware consists of a plug-in board which fits into the PC slot with seven transputers on it. The main programming language for implementing the parallel algorithms on the transputers is 'parallel C'. The overall program consists of a group of subprograms intended to run on different transputers and a configuration file specifying how the subprograms are mapped onto different transputers, and how the links are connected among transputers. 'Parallel C' is augmented with some parallel facilities, including threads (to create parallel processes within a transputer), channels (to communicate through links between two processes on different transputers), and semaphore operations etc.

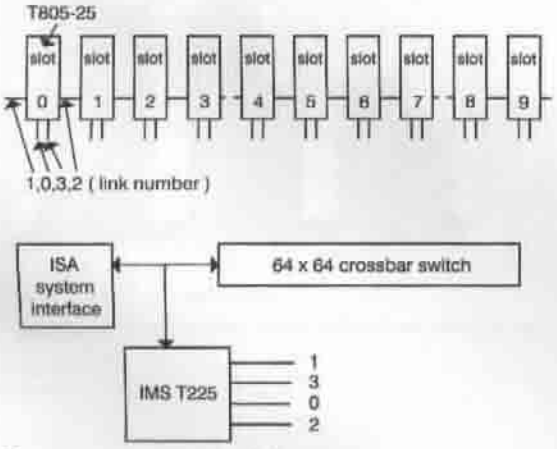

Fig. 5 *Ultra-XL functional block diagram*

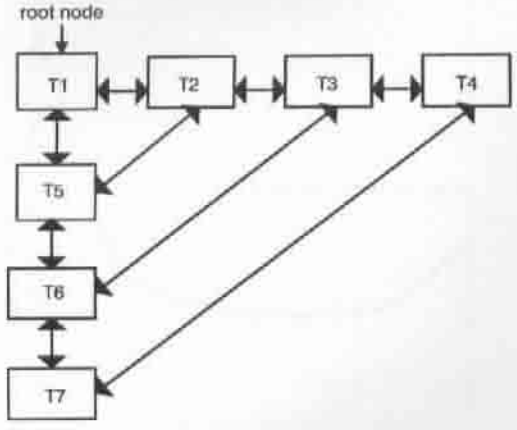

Fig. 6 *Transputer interconnections to parallelise SOLVE efficiently*
T1 computes L and U in diagonal blocks and performs I/O
T2, T3, T4 compute U blocks in the upper-triangular portion of the matrix
T5, T6, T7 compute L blocks in the lower-triangular portion of the matrix
Diagonal connections are used to exchange L, U block values efficiently between transputers needed in different components

Fig. 5 shows the 16-bit add-in card called ULTRA/XL for a 80x86 IBM PC AT bus. The add-in card can contain up to 10 transputer modules (called TRAMs, i.e. a transputer plus 4 Mbyte RAM). Each TRAM is a T805-25 Mhz compute node. The hardware interconnections between the transputers can be changed to some extent by programming the crossbar switch. The communication speed between transputers for the above setup is 2 Mbyte/s. Because the parallel-computing environment is message-passing type, an efficient transputer-interconnection network needs to be set up. An efficient partitioning and mapping scheme was described in Section 2 which maximises concurrency and minimises communication between transputers (see Fig. 1. The transputer interconnection network to implement this scheme is shown in Fig. 6.

4 Results

The performance of the sparse-matrix SOLVE was tested on matrices generated from ISCAS-85 benchmark circuits. The ISCAS-85 circuits are gate-level combinational circuits. These circuits were expanded to their equivalent CMOS transistor-level circuits. From this netlist, the corresponding zero–nonzero structure of the matrix was generated. Table 1 shows the characteristics of the matrices used in this study. Table 2 shows the block-level sparsity in the benchmark matrices for different size blocks.

Table 1: Characteristics of ISCAS-85 benchmark circuits and corresponding matrices

Circuit type	c0432	c0499	c0880	c1355
Transistors	716	1140	1802	2308
Matrix size	360	498	924	1218
Element-level sparsity (%)	99.17	99.36	99.73	99.74

Table 2: Block level sparsity in ISCAS-85 benchmark circuits for different block sizes

Circuit type	c0432	c0499	c0880	c1355
2×2 block sparsity	97.16	98.83	99.22	99.09
3×3 block sparsity	95.45	97.83	98.51	98.36
4×4 block sparsity	92.69	97.51	97.63	96.85
5×5 block sparsity	89.39	96.14	95.25	96.21
6×6 block sparsity	85.69	95.61	95.22	95.02
7×7 block sparsity	82.21	94.72	93.89	93.41
8×8 block sparsity	77.63	95.23	92.39	89.82

The performance of a one-level sparse solution scheme, where the sparsity is exploited at the block level only, was compared with that of a two-level scheme. Both algorithms have a sparse storage scheme using a 2-D pointer array and dynamic allocation of nonzero blocks for memory efficiency, as explained in Section 2.2. The difference between the one- and two-level sparse algorithms is the use of the compressed-sparse-row/column-storage scheme, sparse-block multiply, and sparse-*LU*-decomposition routines to exploit the sparsity within a nonzero block. Thus the two-level sparse algorithm has additional conversion overhead in converting a block stored in dense form to CSR or CSC form. However, this conversion overhead is a minimum since it is $O(m^2)$ where $m \times m$ is the block size. Compare this with a block-matrix-by-block-matrix multiply which is $O(m^3)$. If the sparsity in the block is high, the sparse-multiply time may turn out to be $0.1m^3$ operations (instead of $2m^3$). Thus, even with the conversion overhead, the sparse-block-multiply time will be much less than if the block were treated as dense in the matrix multiplication. The two-level sparse algorithm will be more efficient if block sizes are relatively large so that the sparsity within a nonzero block is higher.

Fig. 7 shows a comparison of execution time among parallel one-level and two-level sparse-matrix *LU*-decomposition times on the c0880 benchmark circuit. For both these cases, a block size of 6×6 was used. By varying the block size for several circuits, it was determined that a size of 6×6 yields low execution times. To show how it compares to the sequential case, the execution time for this is also shown in Fig. 7. It can be seen that two-level sparse algorithm improves the execution time considerably. The parallel sparse algorithms have improvement in execution time as the number of processors is increased. The results obtained are similar for the other benchmark circuits [16].

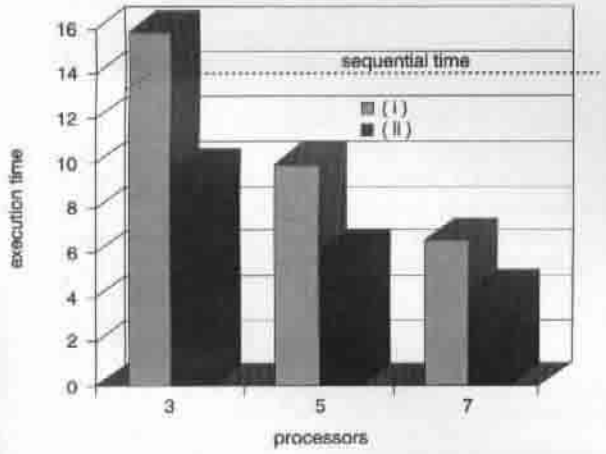

Fig. 7 *Comparison of execution times on the c0880 circuit (matrix size = 924 × 924)*
(i) one-level sparse
(ii) two-level sparse

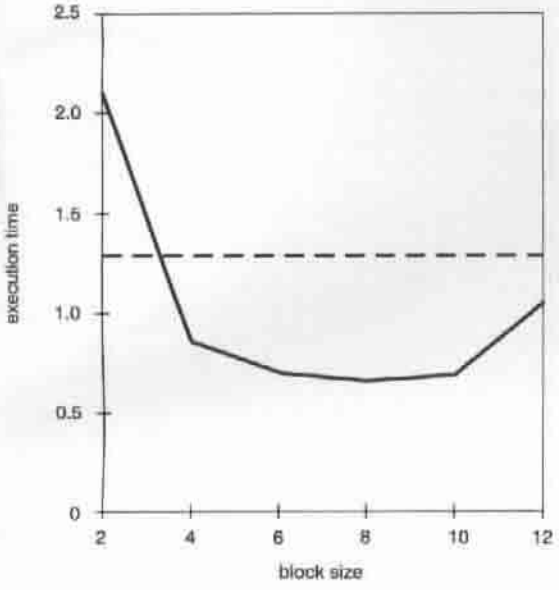

Fig. 8 *Execution time against block size for the two-level sparse algorithm*
——— two-level sparse
– – – – sequential

It is important to determine the partitioned-block size which yields the best execution time. The c0432 benchmark circuit was used for the parallel sparse *LU*-decomposition algorithms to determine the optimum block size for the transputer system. Fig. 8 shows a graph for the two-level sparse algorithm for the c0432 circuit. It indicates that the best execution time is obtained when the block size is from 6×6 to 10×10 with seven processors. For a one-level sparse algorithm, the optimum block size turned out to be between 4×4 and 6×6. This is expected because the one-level sparse algorithm exploits sparsity only at the block level and this sparsity decreases if the block size is increased. The two-level algorithm can tolerate higher block sizes since it can exploit sparsity within a block. The higher block size is needed to match the granularity of the problem, i.e. the computation-to-communication ratio of the processors and the interconnection network employed.

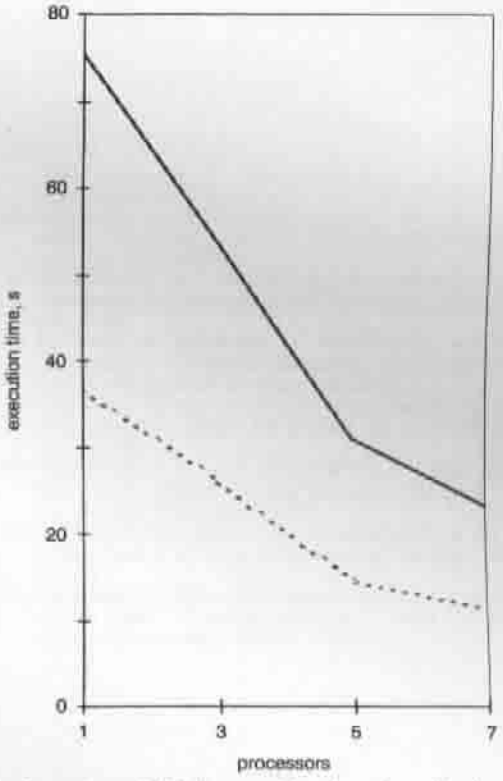

Fig. 9 *Comparison of LU decomposition times for original and reordered matrix*
——— original matrix
– – – – reordered matrix

Fig. 9 shows a comparison of execution time of the original matrix and that of the reordered matrix on the c1355 circuit using the modified Markowitz's algorithm as described in Section 2.3. The execution times are for the parallel two-level sparse-matrix *LU*-decomposition algorithm using a block size of 6×6. It can be seen that the block-level reordering scheme improves the execution time considerably.

5 Conclusions

In this paper, a novel technique which directly partitions the sparse matrix and thus has a better potential for load balancing in a parallel implementation is described. The scheme exploits sparsity both at the block level and within each nonzero block. The two-level sparse scheme further helps the load-balancing problem by minimising the effect of many zero matrix blocks being assigned to a processor during a parallel time step. A novel mapping of matrix blocks to processors has been developed which exploits maximum parallelism in the *LU* decomposition and back substitution according to the number of processors available, and also minimises the communication between processors. A full implementation including the sparse-matrix-stor-

age and block-level-reordering schemes is carried out on a transputer array. Good accelerations are obtained in all benchmarks tested up to the number of processors available for the experiment.

This work needs to be extended to allow for a complete execution environment for a direct simulation program such as SPICE. The present project has implemented only the SOLVE phase. The LOAD phase, which assembles the sparse matrix from a given circuit, is also a time-consuming part which needs to be parallelised. However, the parallelisation of the LOAD part is relatively easy. To maximise overall efficiency, the matrix assembly in the LOAD phase should be divided exactly according to the block assignment for different processors in the SOLVE phase.

6 Acknowledgments

This research was funded in part by a grant from the National Science Foundation Center for the Design of Analog and Digital Integrated Circuits under grant CDADIC 92-4. An initial and shorter version of this paper appeared in the 1996 International Conference on High Performance Computing, HiPC-96.

7 References

1 NAGEL, L.W.: 'SPICE2: a computer program to simulate semiconductor circuits'. Electronics Research Laboratory, UC, Berkeley, memorandum ERL-M520, May 1975

2 COX, P., and BURCH, R.: 'Direct circuit simulation algorithms for parallel processing', *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, 1991, **10**, (6), pp. 714–725

3 SADAYAPPAN, P., and VISVANATHAN, V.: 'Circuit simulation on shared memory multiprocessors', *IEEE Trans.*, 1988, **C–37**, (12), pp. 1634–1642

4 YANG, G.C.: 'PARASPICE: a parallel circuit simulator for shared-memory multiprocessors'. Proceedings of the 27th *Design automation* conference, 1990, pp. 400–405

5 WHITE, J.K., and SANGIOVANNI-VINCENTELLI, A.: 'Relaxation techniques for the simulation of VLSI circuits' (Kluwer Academic Publishers, 1987)

6 TROTTER, J.A., and AGRAWAL, P.: 'Circuit simulation algorithms on a distributed memory multiprocessor system'. Proceedings of *ICCAD-90*, 1990, pp. 438–441

7 SADAYAPPAN, P.: 'Efficient sparse matrix factorization for circuit simulation on vector supercomputers'. Proceedings of the 26th *Design automation* conference, 1989, pp. 13–18

8 VLADIMIRESCU, A., and PEDERSON, D.O.: 'Circuit simulation on vector processors'. Proceedings of the IEEE ICCC, 1982, pp. 172–175

9 GREER, B.: 'Converting SPICE to vector code', *VLSI Syst. Des.*, 1986, **7**, (1)

10 MAHMOOD, A., SPARKS, S., and BAKER, W.I.: 'Systolic algorithms for solving a sparse system of linear equations in circuit simulation', *Integr. VLSI J.*, 1995, **19**, (1), pp. 83–107

11 CHILAKPATI, U.: 'Parallel SOLVE for SPICE on a network of workstations'. MS project report, Washington State University, 1995

12 SANGIOVANNI-VINCENTELLI, A.: 'A new tearing approach-node tearing nodal analysis'. Proceedings of the IEEE international symposium on *Circuits and systems*, 1977, pp. 143–145

13 HWANG, K., and CHENG, Y.H.: 'Partitioned matrix algorithms for VLSI arithmetic systems', *IEEE Trans.*, 1982, **C–31**, (12), pp. 1215–1224

14 NAVARRO, J.J.: 'Partitioning: an essential step in mapping algorithms into systolic array processors', *Computer*, July 1987, pp. 77–89

15 KUMAR, V., GRAMA, A., GUPTA, A., and KARYPIS, G.: 'Introduction to parallel computing, design and analysis of algorithms' (Benjamin/Cummings Publishing Co. Inc., 1994)

16 CHU, Y.: 'Parallel solution of sparse matrix equations in SPICE on a transputer array'. MS project report, Washington State University, 1995

17 TROTTER, J.A., and AGRAWAL, P.: 'Fast overlapped scattered array storage schemes for sparse matrices'. Proceedings of the *ICCAD*, 1990, pp. 450–453

18 MARKOWITZ, H.M.: 'The elimination form of the inverse and its application to linear programming', *Manage. Sci.*, 1957, **3**, pp. 255–269

19 EISENSTAT, S.C.: 'Yale sparse matrix package II: the nonsymmetric codes'. Research report 114, Yale University Computer Science Department, 1977