INTERNATIONAL JOURNAL OF NUMERICAL MODELLING : ELECTRONIC NETWORKS, DEVICES AND FIELDS,
Vol. 7, 329–342 (1994)

# PARALLEL COMPUTATION OF 3-D ELECTROMAGNETIC SCATTERING USING FINITE ELEMENTS

A. CHATTERJEE,[1] J. L. VOLAKIS[1] AND D. WINDHEISER[2]

*Radiation Laboratory[1] and Advanced Computer Architecture Laboratory,[2] Department of Electrical Engineering and Computer Science, University of Michigan, Ann Arbor, MI 48109-2122, U.S.A.*

## SUMMARY

The finite element method (FEM) with local absorbing boundary conditions has been recently applied to compute electromagnetic scattering from large 3-D geometries. In this paper, we present details pertaining to code implementation and optimization. Various types of sparse matrix storage schemes are discussed and their performance is examined in terms of vectorization and net storage requirements. The system of linear equations is solved using a preconditioned biconjugate gradient (BCG) algorithm and a fairly detailed study of existing point and block preconditioners (diagonal and incomplete LU) is carried out. A modified ILU preconditioning scheme is also introduced which works better than the traditional version for our matrix systems. The parallelization of the iterative sparse solver and the matrix generation/assembly as implemented on the KSR1 multiprocessor is described and the interprocessor communication patterns are analysed in detail. Near-linear speed-up is obtained for both the iterative solver and the matrix generation/assembly phases. Results are presented for a problem having 224,476 unknowns and validated by comparison with measured data.

## 1. INTRODUCTION

Differential equation techniques are rapidly becoming the preferred solution methods for the computation of electromagnetic scattering and radiation from inhomogeneous, three-dimensional geometries.[1,2] In the finite element method, the computational domain is at first discretized using node-based or edge-based finite elements. Edge-based elements are more desirable for representing electromagnetic fields because they exhibit tangential continuity and normal discontinuity across interelement boundaries and material discontinuities. Moreover, they can treat geometries with sharp edges and are divergenceless. The outer boundary of the finite element mesh is artificially truncated at some distance from the target using an absorbing boundary condition (ABC). ABCs are essentially differential equations chosen to suppress non-physical reflections from the boundary, thus ensuring the outgoing nature of the waves. They are approximate boundary conditions but have the important advantage of retaining the sparsity of the matrix system which leads to an $O(N)$ storage requirement. The number of unknowns is further reduced since it is found that reliable results are obtained by truncating the artificial boundary only a fraction of a wavelength from the target.

In our FE-ABC implementation, we use an edge-based finite element formulation coupled with vector ABCs on conformal boundaries to compute scattering from three-dimensional structures having regions satisfying impedance and/or transition conditions. The limiting factor in dealing with three-dimensional geometries is usually the number of unknowns and the corresponding demands on storage and solution time. Solution techniques which have $O(N)$ storage and feasible solution time are, therefore, the only way that three-dimensional problems can be solved with the available computer resources. This is one of the principal reasons for the popularity of partial differential equation techniques over integral equation (IE) approaches which in contrast lead to dense matrices. As the problem size increases, the IE and hybrid methods (both need $O(N^l)$, $1 < l \leqslant 2$ storage) quickly become unmanageable in terms of storage and solution time. Another concern while solving problems having more than 100,000 unknowns—a scenario that can be envisioned for most practical problems—is to avoid software bottlenecks. The algorithmic complexity of any part of the program should increase at most linearly with the number of unknowns.

In this paper, the implementation details of our finite element code are presented along with

CCC 0894-3370/94/050329-14 *Received 4 August 1993*

the associated numerical considerations. The various tradeoffs associated with the data structures used to represent sparse matrices and their impact on vectorization and parallelization are discussed. The iterative solver—a preconditioned biconjugate gradient (BCG) algorithm—is studied along with point and block preconditioning strategies and the tradeoffs between the two types of preconditioners are outlined. A modified incomplete LU (ILU) preconditioner is presented, which seems to work better than the original ILU preconditioner for our matrix systems. The computationally intensive portions of the finite element code have been parallelized on the KSR1 (Kendall Square Research) shared-address space distributed-cache architecture with substantial speed-up. A full analysis of the communication patterns is presented and the solution methodology is validated by comparison with measured data.

## 2. FORMULATION

We consider the problem of scattering from an inhomogeneous geometry with material discontinuities. The scatterer is enclosed within a fictitious surface, denoted by $S_o$, where the ABCs are applied. The second-order vector ABC is given by Reference 3

$$\hat{\mathbf{n}} \times \nabla \times \mathbf{E}^s = \alpha \mathbf{E}_t^s + \beta \nabla \times [\hat{\mathbf{n}}(\nabla \times \mathbf{E}^s)_n] + \beta \nabla_t (\nabla \cdot \mathbf{E}_t^s) \tag{1}$$

where $\alpha = jk$, $\beta = 1/(2jk + 2/r)$, $\mathbf{E}^s$ represents the scattered electric field, $\hat{\mathbf{n}}$ is the unit normal to the surface and the subscripts t and n denote the transverse and normal component to $S_o$, respectively. Inside the volume, $V$, the scattered field satisfies the Helmholtz vector wave equation and boundary conditions associated with the material properties of the body. A detailed formulation of these boundary conditions has been given in Reference 1. The functional involving the scattered electric field ($\mathbf{E}^s$) to be discretized in connection with our proposed FE-ABC formulation is given by

$$\begin{aligned} F(\mathbf{E}^s) = & \int_V \left[ \frac{1}{\mu_r} (\nabla \times \mathbf{E}^s) \cdot (\nabla \times \mathbf{E}^s) - k_o^2 \epsilon_r \mathbf{E}^s \cdot \mathbf{E}^s \right] dV \\ & + jk_o Z_o \int_{S_k} \frac{1}{K} (\hat{\mathbf{n}} \times \mathbf{E}^s) \cdot (\hat{\mathbf{n}} \times \mathbf{E}^s) dS \\ & + \int_{S_o} \mathbf{E}^s \cdot P(\mathbf{E}^s) dS \\ & + 2jk_o Z_o \int_{S_d} \frac{1}{\mu_r} \mathbf{E}^s \cdot (\hat{\mathbf{n}} \times \mathbf{H}^{inc}) dS \\ & + 2 \int_{V_d} \left[ \frac{1}{\mu_r} (\nabla \times \mathbf{E}^s) \cdot (\nabla \times \mathbf{E}^{inc}) - k_o^2 \epsilon_r \mathbf{E}^s \cdot \mathbf{E}^{inc} \right] dV \\ & + 2jk_o Z_o \int_{S_k} \frac{1}{K} (\hat{\mathbf{n}} \times \mathbf{E}^s) \cdot (\hat{\mathbf{n}} \times \mathbf{E}^{inc}) dS \\ & + f(\mathbf{E}^{inc}) \end{aligned} \tag{2}$$

where $\epsilon_r$ and $\mu_r$ are the respective relative permittivity and permeability of the dielectric materials, $V_d$ is the volume occupied by the dielectric (portion of $V$ where $\epsilon_r$ or $\mu_r$ are not unity), $S_d$ encompasses all dieletric interface surfaces and $S_k$ represents the surface of a resistive (or an impedance sheet) with resistivity (or impedance) $K$. $\mathbf{E}^{inc}$ is the incident plane wave given by

$$\mathbf{E}^{inc}(\mathbf{r}) = [(\hat{\boldsymbol{\alpha}} \cdot \hat{\boldsymbol{\theta}}^i)\hat{\boldsymbol{\theta}}^i + (\hat{\boldsymbol{\alpha}} \cdot \hat{\boldsymbol{\phi}}^i)\hat{\boldsymbol{\phi}}^i] e^{-j\mathbf{k}^i \cdot \mathbf{r}} \tag{3}$$

where $\hat{\boldsymbol{\alpha}} = \hat{\boldsymbol{\theta}}^i \cos\alpha + \hat{\boldsymbol{\phi}}^i \sin\alpha$ is the polarization vector, $\mathbf{k}^i$ is the propagation vector

$$\mathbf{k}^i = -k_o(\sin\theta^i\cos\phi^i\hat{\mathbf{x}} + \sin\theta^i\sin\phi^i\hat{\mathbf{y}} + \cos\theta^i\hat{\mathbf{z}}) \tag{4}$$

and $\hat{\boldsymbol{\theta}}^i$, $\hat{\boldsymbol{\phi}}^i$ are the usual unit vectors in the spherical co-ordinate system. $f(\mathbf{E}^{inc})$ is a function of the incident electric field only and vanishes on differentiating the functional.

## 3. FINITE ELEMENT DISCRETIZATION

To discretize (2), the computational volume $V$ is subdivided into a number of small tetrahedra, each occupying the volume $V^e$ ($e = 1, 2, \ldots, M$), where $M$ denotes the total number of tetrahedral elements. Within each element, the scattered electric field is expressed as

$$\mathbf{E}^e = \sum_{j=1}^{m} E_j^e \mathbf{W}_j^e = \{\mathbf{W}^e\}^T\{E^e\} = \{E^e\}^T\{\mathbf{W}^e\} \tag{5}$$

where $\mathbf{W}_j^e$ are the edge-based vector basis functions,[4] $E_j^e$ denote the expansion coefficients of the basis and represent the field components tangential to the $j$th edge of the $e$th element, $m$ is the number of edges making up the element and the superscript stands for the element number. The basis functions used in our implementation have zero divergence and constant curl.

The system of equations to be solved for $E_j^e$ is obtained by a Rayleigh–Ritz procedure which amounts to differentiating $F(\mathbf{E}^s)$ with respect to each edge field and then setting it to zero. On substituting the basis expansion into the expression for the functional, taking the first variation in $F(\mathbf{E}^s)$ and assembling all $M$ elements, we obtain the following augmented system of equations

$$\left\{\frac{\partial F}{\partial E^e}\right\} = \sum_{e=1}^{M}[A^e]\{E^e\} + \sum_{s=1}^{M_s}[B^s]\{E^s\} + \sum_{p=1}^{M_p}\{C^p\} = 0 \tag{6}$$

In this, $M_s$ denotes the number of triangular surface elements on $S_k$ and $S_o$ whereas $M_p$ is equal to the sum of the surface elements on $S_k$, $S_d$ and the volume elements in $V_d$. The elements of the matrices $[A^e]$, $[B^s]$ and $\{C^p\}$ are given in Reference 1. The final system can be expressed as

$$[\mathbf{A}]\{\mathbf{x}\} = \{\mathbf{b}\} \tag{7}$$

where $\{\mathbf{x}\}$ is the unknown vector representing the weighting coefficients of the basis functions.

The imposition of boundary conditions on the finite element mesh is usually quite simple. No special treatment is required at material discontinuities; the mere identification of surface elements lying on material discontinuities or inhomogeneities kicks in the contribution from the surface integrals in $F(\mathbf{E}^s)$. For perfectly conducting scatterers, the interior region is not meshed since the electromagnetic wave does not penetrate inside the scatterer. If the surface element lies on a metallic boundary, a simple modification is carried out on the element matrix to preserve the symmetry of the matrix system.

## 4. NUMERICAL CONSIDERATIONS

The finite element code implemented by the authors can be divided into four main modules:

- Input/output
- Right-hand side vector (**b**) generation
- Finite element matrix (**A**) generation
- Linear equation solver

The input to the program consists of the mesh information obtained by preprocessing the mesh file generated from SDRC I-DEAS, a commercial CAD software package. The right-hand side vector (**b**) is usually a sparse vector and only a small fraction of the total CPU time is required to generate it. The finite element matrix generation consists of too many subroutine calls and highly complex loops to permit any significant speed-up through vectorization. It is, however, highly amenable to parallelization as will be discussed later. The most time-consuming portion

of the code is the linear equation solver taking up approximately 90 per cent of the CPU time. On a vector computer like the Cray YMP, it is possible to vectorize only the equation solver. However, short vector lengths and indirect addressing inhibit large vector speed-ups.

### 4.1. *Matrix generation*

The matrix systems arising from I-DEAS were very sparse: on the average, the minimum number of non-zero elements per row was 9 and the maximum number of non-zeros per row was 30. The total number of non-zeros varied between $15N$ and $16N$, where $N$ is the number of unknowns.

There are various storage schemes for sparse matrices. In this paper, we will discuss the ITPACK format[5] and the Compressed Sparse Row (CSR) format. The ITPACK storage scheme is attractive for generating finite element matrices since the number of comparisons required while augmenting the matrix depends only on the locality of the corresponding edge and not on the number of unknowns. Moreover, the sparse matrix–vector multiplication process can be highly vectorized when the number of non-zeros in all rows is nearly equal. However, for our application, almost half the space is lost in storing zeros. The modified ITPACK scheme[6] does alleviate this problem to a certain degree by sorting the rows of the matrix and decreasing the number of non-zero elements. However, 30% of the allotted space is still lost in zero padding. The best tradeoff between storage and speed for our application is obtained by storing the non-zero matrix elements in a long complex vector, the column indices in a long integer vector and the number of non-zeros per row in another integer vector. This data structure is referred to as the compressed sparse row (CSR) format. In our implementation, a map of the number of non-zeros for each row is obtained through a simple preprocessor. The main program stores the matrix in CSR format, thus minimizing storage and sacrificing a bit of speed. The required storage is $15N$ complex words plus integers for $X$ and $PC$, respectively, and $N$ integers for the array containing the pointers to the rows' data.

### 4.2. *Linear equation solver*

In three-dimensional applications, the order $N$ of the system of linear equations may be very large. Direct solution methods usually suffer from fill-in to an extent that these large problems cannot be solved at a reasonable cost even on state-of-the-art parallel machines. It is, therefore, essential to employ solvers whose memory requirements are a small fraction of the storage demand of the coefficient matrix. This necessitates the use of iterative algorithms instead of direct solvers to preserve the sparsity pattern of the finite element matrix. Especially attractive are iterative methods that involve the coefficient matrices only in terms of matrix–vector products with $\mathbf{A}$ or $\mathbf{A}^T$. The most powerful iterative algorithm of this type is the conjugate gradient algorithm for solving positive definite linear systems.[7] In our implementation, the system of linear equations is solved by a variation of the CG algorithm, the biconjugate gradient (BCG) method. This scheme is usually used for solving unsymmetric systems; however, it performs equally well when applied to symmetric systems of linear equations. For symmetric matrices, BCG differs from CG in the way the inner product of the vectors are taken. The conjugate gradient squared (CGS) algorithm[8] is usually faster than BCG but is more unstable since the residual polynomials are merely the squared BCG polynomials and hence exhibit even more erratic behaviour than the BCG residuals. Moreover, there are cases where CGS diverges, while BCG still converges. Recently, Freund[9] has proposed the quasi-minimal residual (QMR) algorithm with look-ahead for complex symmetric matrices.

Based on the above, the biconjugate gradient (BCG) algorithm was found to be most suitable for our implementation. The BCG requires one matrix–vector multiplication, three vector updates and three dot products per iteration. The solution scheme requires only three additional vectors of length $N$. The vector updates and the dot products can be carried out extremely quickly on a vector Cray machine such as the Cray YMP, reaching speeds of about 190 MFLOPS. However, the matrix–vector product, which involves indirect addressing and short vector lengths, runs at about 45·5 MFLOPs on one processor of the eight-processor Cray YMP. As a rule of thumb,

the biconjugate gradient algorithm with no preconditioning consumes 4·06 microseconds per iteration per unknown on the Cray YMP.

### 4.3. *Preconditioning*

The condition number of the system of equations usually increases with the number of unknowns. It is then desirable to precondition the coefficient matrix such that the modified system is well-conditioned and converges in significantly fewer iterations than the original system. The equivalent preconditioned system is of the form

$$[\mathbf{C}^{-1}][\mathbf{A}]\{\mathbf{x}\} = [\mathbf{C}^{-1}]\{\mathbf{b}\} \tag{8}$$

The non-singular preconditioning matrix **C** must satisfy the following conditions:

(1) should be a good approximation to **A**.
(2) should be easy to compute.
(3) should be invertible in $O(N)$ operations.

The preconditioners that we discuss below are the diagonal and the ILU point and block preconditioners. Block preconditioners are usually preferable, owing to reduced data movement between memory level hierarchies as well as decreased number of iterations required for convergence. Block algorithms are also suited for high-performance computers with multiple processors since all scalar, vector and matrix operations can be performed with a high degree of parallelism.

4.3.1. *Diagonal preconditioner.* The simplest preconditioner that was used in our implementation was the point diagonal preconditioner. The preconditioning matrix **C** is a diagonal matrix which is easy to invert and has a storage requirement of $N$ complex words, where $N$ is the number of unknowns. The entries of **C** are given by

$$C_{ij} = \delta_{ij} A_{ij}, \qquad i = 1, \ldots, N; \qquad j = 1, \ldots, N \tag{9}$$

where $\delta_{ij}$ is the Kronecker delta. The matrix $\mathbf{C}^{-1}$ contains the reciprocal of the diagonal elements of **A**. The algorithm with the diagonal preconditioner converged in about 35 per cent of the number of iterations required for the unpreconditioned case. This suggested that our finite element matrix was diagonally dominant since the reduction in the number of iterations was rather impressive. The diagonal preconditioner is also easily vectorizable and consumes 4·1 microseconds per iteration per unknown on the Cray YMP, a marginal slowdown over the unpreconditioned system.

A more general diagonal preconditioner is the block-diagonal preconditioner. The point-diagonal preconditioner is a block-diagonal preconditioner with block size 1. The block-diagonal preconditioning matrix consists of $m \times m$ symmetric blocks as shown in Figure 2. The inverse of the whole matrix is simply the inverse of each individual block put together. If the preconditioning matrix **C** is broken up into $n$ blocks of size $m$, the storage requirement for the preconditioner is at most $m \times N$. However, this method suffers a bit from fill-in since the inverted $m \times m$ blocks are dense even though the original blocks may have been sparse. For this reason, large blocks cannot be created since the inverted blocks would lead to full matrices and take a significant fraction of the total CPU time for inversion. However, since the structure of the preconditioning matrix is known *a priori*, this preconditioner vectorizes well and runs at 194 MFLOPS (line 5 of Figure 1) on the Cray-YMP for a block size of 8. For a test case of 20,033 unknowns, a block size of 2 caused the maximum reduction in the number of iterations (14 per cent) and ran at 197 MFLOPS.

4.3.2. *Modified ILU preconditioner.* The next step was to use a better preconditioner to improve the condition number of the system resulting in faster convergence. The traditional ILU preconditioner[10] was employed with zero fill-in; however, the algorithm took a greater number of iterations than the diagonal preconditioner to converge to a specified tolerance. This was probably because the ILU preconditioned system may not have been positive definite.[11] The preconditioned conjugate gradient method usually converges faster if the preconditioner is positive

Initialization:

$\boldsymbol{x}$ given
$\boldsymbol{r} = \boldsymbol{b} - A\boldsymbol{x}$
$\boldsymbol{p} = \boldsymbol{r}$
tmp $= \boldsymbol{r} \cdot \boldsymbol{r}$

Repeat until $(resd \leq tol)$

$$\left.\begin{array}{ll} \boldsymbol{q} = A\boldsymbol{p} & (1) \\ \alpha = tmp/(\boldsymbol{q} \cdot \boldsymbol{p}) & (2) \end{array}\right\} \text{ Step 1}$$

$$\left.\begin{array}{ll} \boldsymbol{x} = \boldsymbol{x} + \alpha\boldsymbol{p} & (3) \\ \boldsymbol{r} = \boldsymbol{r} - \alpha\boldsymbol{q} & (4) \\ \boldsymbol{q} = \mathbf{C}^{-1} * \boldsymbol{r} & (5) \\ resd = \sqrt{|\boldsymbol{r} \cdot \boldsymbol{r}^*|} & (6) \\ \beta = (\boldsymbol{r} \cdot \boldsymbol{q})/tmp & (7) \end{array}\right\} \text{ Step 2}$$

$$\left.\begin{array}{ll} tmp = \beta \times tmp & (8) \\ \boldsymbol{p} = \boldsymbol{q} + \beta\boldsymbol{p} & (9) \end{array}\right\} \text{ Step 3}$$

EndRepeat

$A$ is a sparse complex symmetric matrix.
$C$ is the preconditioning matrix.
$\boldsymbol{q},\boldsymbol{p},\boldsymbol{x},\boldsymbol{r}$ are complex vectors.
$\alpha,\beta,tmp$ are complex scalars.
$resd,tol$ are real scalars.

Figure 1. Symmetric biconjugate gradient method with preconditioning

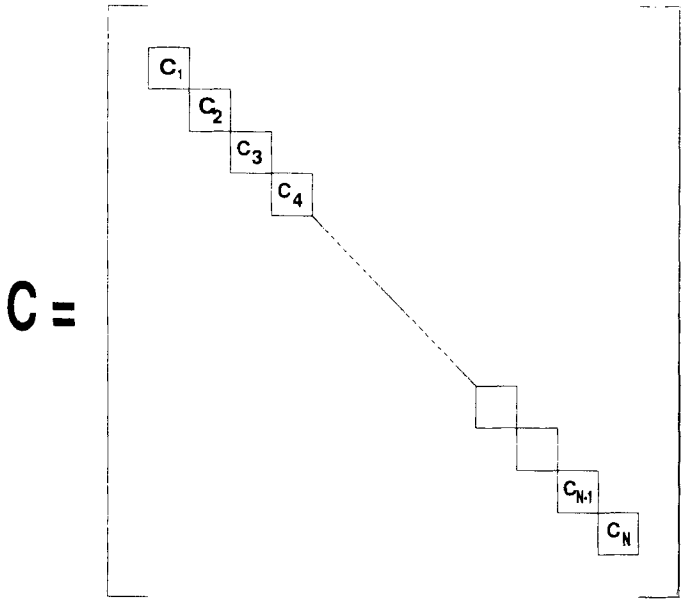



Figure 2. Structure of block preconditioning matrix

definite, although this is not a necessary condition. Higher values of fill-in were not attempted since the preconditioner already occupied space equal to that of the coefficient matrix.

A modified version of the ILU preconditioner was next employed by eliminating the inner loop of the traditional version. The algorithm is outlined in the Appendix and basically scales the off-diagonal elements in the lower triangular portion of the matrix by the column diagonal. Since

the matrix is symmetric, it retains the $LDL^T$ form and is also positive definite if the coefficient matrix is positive definite. This preconditioner is less expensive to generate and converges in about one-third of the number of iterations taken by the point-diagonal preconditioner. It has been tested with reliable results for $N \leq 50{,}000$. However, the time taken by the two preconditioning strategies is approximately the same since each iteration of the ILU preconditioned system is about three times more expensive. The forward and backward substitutions carried out at each iteration runs at 26·5 MFLOPS on the Cray YMP and proves to be the bottleneck since they are inherently sequential processes and the vector lengths are approximately half that of the sparse matrix–vector multiplication process. The triangular solver is also extremely difficult to parallelize. Techniques like level scheduling and self-scheduling have been used to exploit the fine grain parallelism in the sparse system but without much success.[12]

As with the diagonal preconditioner, a block version of the ILU preconditioner was also attempted. This strategy distributes one block to each processor in a multiprocessor architecture thus achieving load balancing as well as minimizing fill-in. The modified ILU decomposition outlined earlier is then carried out on each of these individual blocks. Further, since the blocks are much larger than the block-diagonal version, the preconditioner is a closer approximation to the coefficient matrix. Moreover, the triangular solver is fully parallelized since each processor solves an independent system of equations through forward and backward substitution. In our test case of 20,033 unknowns, the number of iterations was reduced by approximately half the number required by the diagonal preconditioner. Since the work done is less than twice that for the diagonal preconditioner, we achieved a marginal savings of CPU time. However, the number of iterations required for convergence is highly sensitive to block size as shown in Table I for $N = 20{,}033$. Table I clearly shows that a larger block size (smaller number of blocks) does not guarantee faster convergence. However, there is an approximately 50 per cent decrease in the number of iterations over the point-diagonal preconditioner, regardless of block size. The optimum block size is dependent on the sparsity pattern of the matrix and can only be determined empirically. The savings in the number of iterations over the point diagonal preconditioner for 28 blocks is given in Table II for a system having 224,476 unknowns.

From the table, it is clear that the block ILU preconditioner is very effective in reducing the iteration count; however, the CPU time required is about 10 per cent less than that required by the point-diagonal preconditioner for the best case.

### 4.4. *Parallelization*

The different versions of the FE-ABC code were parallelized on a KSR1 massively parallel machine which implements a shared virtual memory, although the memory is physically distributed for the sake of scalability. The basic strategy for the parallelization of the code is described on the biconjugate gradient solver with diagonal preconditioning. The other versions use the same parallelization scheme with slight modifications. We also comment on the parallelization of the matrix assembly phase.

The symmetric biconjugate gradient method iteratively refines an approximate solution of the given linear system until convergence. Figure 1 shows the method in terms of vector and matrix

Table I. Number of iterations versus number of blocks for a block ILU preconditioned biconjugate gradient solution method

| No. of blocks | No. of iterations |
|---|---|
| 1 | 127 |
| 2 | 176 |
| 4 | 185 |
| 8 | 172 |
| 12 | 162 |
| 16 | 174 |
| 24 | 223 |
| 28 | 177 |

Table II. Number of iterations required for convergence of a 224,476 unknown system using the point-diagonal and block ILU preconditioning strategies

| Angle of incidence | No. of iterations | | Ratio (II/I) |
|---|---|---|---|
| | point diagonal (I) | block ILU (II) | |
| 0 | 2943 | 2758 | 0·937 |
| 10 | 5985 | 3834 | 0·641 |
| 20 | 5464 | 3984 | 0·729 |
| 30 | 6048 | 3651 | 0·604 |
| 40 | 5770 | 3256 | 0·564 |
| 50 | 5107 | 3720 | 0·728 |
| 60 | 6517 | 4162 | 0·639 |
| 70 | 5076 | 4108 | 0·809 |
| 80 | 5305 | 3551 | 0·669 |
| 90 | 2898 | 2832 | 0·977 |

operations. For a system of equations containing $N$ unknowns, all these vectors are of size $N$ and the sparse matrix is of order $N$. The number of non-zero elements in the sparse matrix is denoted as *nze*. Table III shows the operation count per iteration for each type of vector operation. In the FE-ABC code, each vector operation is implemented as a loop. The program is parallelized by tiling these loops. For $P$ processors, the vectors are divided into $P$ sections of $N/P$ consecutive elements. Each processor is assigned the same section of each vector. This partitioning attempts to reduce communication while balancing load. To guarantee correctness, synchronization points are added after lines 2, 7, and 9. Lines 2 and 7 require synchronization to guarantee that the dot products are computed correctly. Note that the dot products in lines 6 and 7 require only one synchronization. The line 9 synchronization guarantees that $\boldsymbol{p}$ is completely updated before the matrix multiply for the next iteration begins.

In the sparse matrix–vector multiplication, each processor computes a block of the result vector by multiplying the corresponding block of rows of the sparse matrix with the operand vector. Since the operand vector is distributed among the processors, data communication is required. The communication pattern is determined by the sparsity structure of the matrix, which in our case is derived from an unstructured mesh. Therefore the communication pattern is unstructured and irregular. However, since the sparse matrix is not modified during the iterative process, the communication pattern is the same at each iteration. Vector updates and dot products are easily parallelized using the same block distribution as in the sparse matrix vector multiply.

Although sparse computations are known to be hard to implement efficiently on distributed memory machine, mainly because of the unstructured and irregular communication, the previous scheme was easily and efficiently implemented on the KSR1 MPP thanks to the global address space.[13] Table IV shows the execution time of one iteration (in seconds) and the speed-up for different numbers of processors and for two problem sizes.

For both problems, the performance scales surprisingly well up to a large number of processors. For the 20,033-unknown problem, the speed-up for the parallelized sparse solver varies from 1 to 19 as the number of processors is increased from 1 to 28 (Figure 3). The overall performance of the solver on 28 processors is more than three times that of a single processor on the Cray-YMP. The large problem (224,476 unknowns) exhibits superlinear speed-up which can be attributed to a memory effect. As a matter of fact, the large data set does not entirely fit in the local cache of a single node in the KSR which results in a large number of page faults. However, as the

Table III. Floating point operations per iteration

| Operation | Complex | | Real | |
|---|---|---|---|---|
| | * | + | * | + |
| Matrix multiply | *nze* | *nze* − *N* | 4*nze* | 4*nze* − 2*N* |
| Vector updates | 4*N* | 3*N* | 16*N* | 12*N* |
| Dot products | 3*N* | 3*N* | 12*N* | 12*N* |

Table IV. Execution time and speed-up for the iterative solver

| Procs | $N$ = 20,033 | | $N$ = 224,476 | |
|---|---|---|---|---|
| | Execution time (seconds per iteration) | Speed-up | Execution time (seconds per iteration) | Speed-up |
| 1[a] | 0·515 | 1 | 10·8 | 1 |
| 8 | 0·071 | 7·3 | 1·4 | 7·7 |
| 16 | 0·040 | 12·9 | 0·671 | 16·1 |
| 29 | 0·027 | 19·1 | 0·304 | 35·6 |
| 60[b] | | | 0·149 | 76·2 |

[a]For 1, 8 and 16 processors, only the first 100 iterations were run.
[b]Code run on a 64 node KSR at Cornell University.

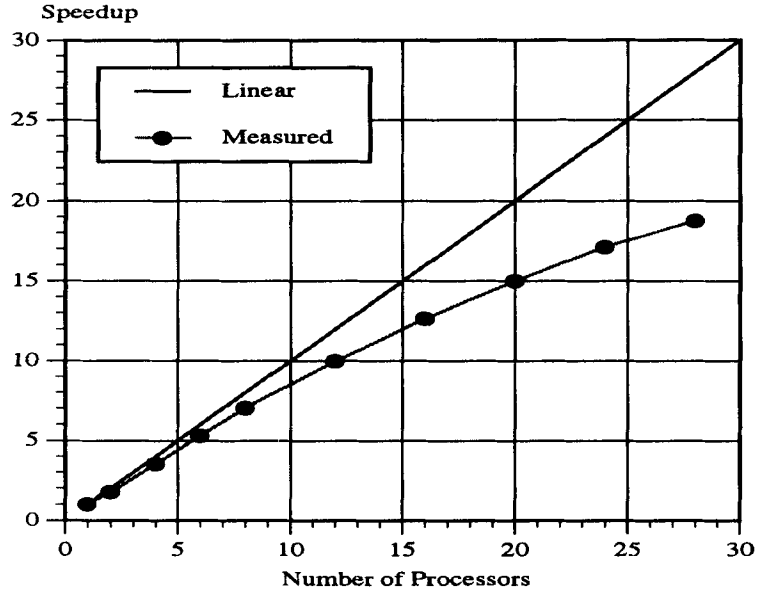



Figure 3. Speed-up curve for the linear equation solver on the KSR1

number of processors increases, the large data set is distributed over the different processors' memories.

The global matrix assembly is the second largest computation in terms of execution time. The elemental matrices are computed for each element in the 3-D mesh and assembled in a global sparse matrix. A natural way of parallelizing the global matrix assembly is to distribute the elements over the processors, have each processor compute the elemental matrix of the elements it owns and update the global sparse matrix. Since the global sparse matrix is shared by all processors, the update needs to be done atomically. On the KSR1 this is done by using the hardware lock mechanism. The performance for the matrix assembly is given in Table V.

4.4.1. *Analysis of communication.* In the main loop (Figure 1), significant communication between processors takes place only during the sparse matrix vector multiply (line 1) and the vector update of $\boldsymbol{p}$ (line 9). The rest of the vector operations incur little or no communication at all. The distribution of the non-zero entries in the matrix affects the amount and nature of communication. In this section, we present an analysis of the communication pattern incurred by the sparse matrix vector multiplication as derived from analysis of the sparsity structure of the matrix.

*Line 1.* In the matrix–vector multiply, each processor computes an $N/P$-sized subsection of the product $\boldsymbol{q}$. The processor needs the elements of $\boldsymbol{p}$ that correspond to the non-zero elements found in the $N/P$ rows of $A$ that are aligned with its subsection. Because the matrix $A$ remains constant throughout the program, the set of elements of $\boldsymbol{p}$ that a given processor needs is the same for

Table V. Execution time and speed-up for the matrix generation and assembly (20,033 unknowns)

| Procs | Execution time in seconds | Speed-up |
|---|---|---|
| 1 | 24·355 | 1 |
| 2 | 13·376 | 1·8 |
| 4 | 6·811 | 3·6 |
| 8 | 3·744 | 6·5 |
| 16 | 1·89 | 12·9 |
| 25 | 1·625 | 15·0 |
| 28 | 1·276 | 19·1 |

all iterations in the loop. However, since $\boldsymbol{p}$ is updated at the end of each iteration, all copies of its element set are invalidated in each processor's local cache except for the ones that the processor itself updates. As a result, in each iteration, processors must obtain updated copies of the required elements of $\boldsymbol{p}$ that they do not own.

These elements can be updated by a read miss to the corresponding subpage, by an automatic update, or by an explicit prefetch or poststore instruction. Figure 4 lists the number of subpages that each of the 28 processor needs to acquire from other processors. Automatic update of an invalid copy of a subpage becomes more likely as the number of processors sharing this subpage grows. The number of processors that need a given subpage (excluding the processor that updates the subpage) is referred to as the *degree of sharing* of that subpage. Figure 5 shows the degree of sharing histogram for the example problem. Since the only subpage misses occuring in Step 1 of the sparse solver are coherence misses due to the vector $\boldsymbol{p}$, the use of the poststore instruction to broadcast the updated sections of the vector $\boldsymbol{p}$ from step 3 should eliminate the subpage misses in step 1. However, the overhead of executing the poststore instruction in step 3 offsets the reduction in execution time of step 1. On a poststore, the processor typically stalls for 32 cycles while the local cache is busy for 48 cycles. As a result, the net reduction in execution time is only 3 per cent.

*Line 9.* Before proceeding with the updates of the $N/P$ elements of $\boldsymbol{p}$ for which it is responsible, each processor must acquire exclusive ownership for those elements. Because a cache line holds eight consecutive elements, each processor will generate $N/8P$ requests for ownership (assuming

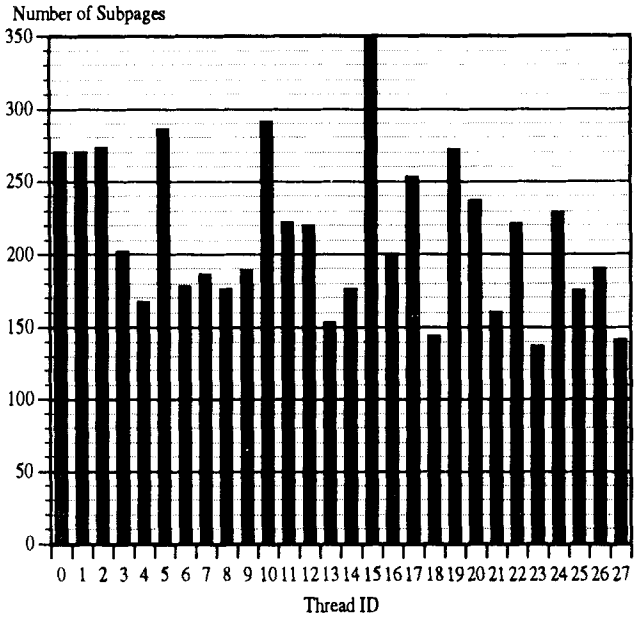



Figure 4. Counts of $\boldsymbol{p}$ subpages required by each processor for sparse matrix–vector multiply (total copies = 5968)

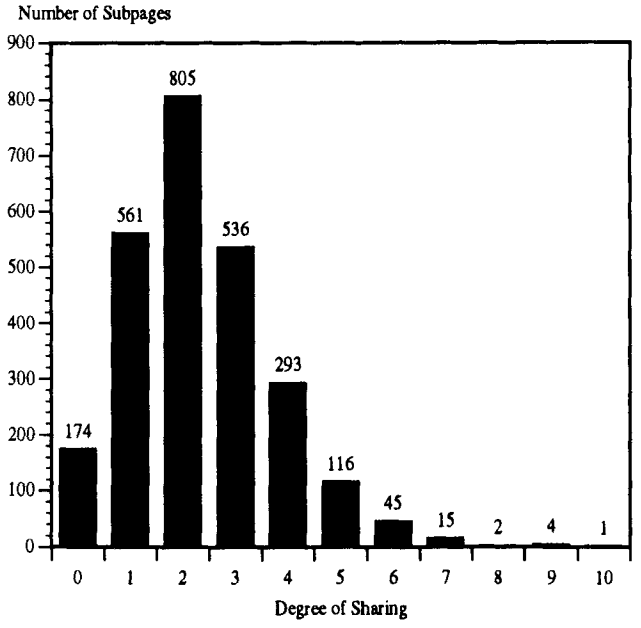



Figure 5. Degree of sharing histogram of $p$ subpages during sparse matrix–vector multiply (28 processors)

all subpages are shared). In order to hide access latencies, the request for ownership can be issued in the form of a prefetch instruction after step 1. This could lead to an eightfold decrease in the number of subpage misses. However, as with the poststore instruction, the benefit of prefetching is offset by the overhead of processing the prefetch instructions in step 2. This is because the processor stalls for at least two cycles on prefetch and the local cache cannot satisfy any processor request until the prefetch is put on the ring. The overall execution time is reduced by only 4 per cent in this case.

*Lines 2, 6, 7.* The rest of the communication is due to the three dot products. Each processor computes the dot product for the vector subsection that it owns. These are then gathered and summed up on a single processor.

## 5. RESULTS

The parallelized code was run for a 1·5λ × 1λ × 1λ (see inset of Figure 6) perfectly conducting rectangular inlet and the radar cross-section computed for both polarizations of the incident plane wave. The radar cross-section (RCS) of a target is given by

$$\sigma = \lim_{r \to \infty} 4\pi r^2 \frac{|\mathbf{E}^s(\mathbf{r})|^2}{|\mathbf{E}^{inc}(\mathbf{r})|^2}$$

The backscatter RCS is obtained when the angle of incidence is the same as the angle of observation (i.e., $\theta^o = \theta^i$, $\phi^o = \phi^i$). In Figure 6, the θ-component of the scattered field is used to compute the backscatter RCS ($\sigma_{\theta\theta}$) of the inlet geometry from a θ-polarized incident wave (i.e., $\alpha = 0°$) taken in the $yz$ plane and compared with measured data.[14] In Figure 7, the backscatter pattern ($\sigma_{\phi\phi}$) from a φ-polarized incident wave is compared to measured data.[14] For the results shown in Figures 6 and 7, the ABC was enforced on a sphere of radius 1·35λ. The discretized geometry had 224,476 unknowns and converged in an average of 3600 iterations on the KSR1 when using the block ILU preconditioner. The agreement is indeed quite good over the entire angular range.

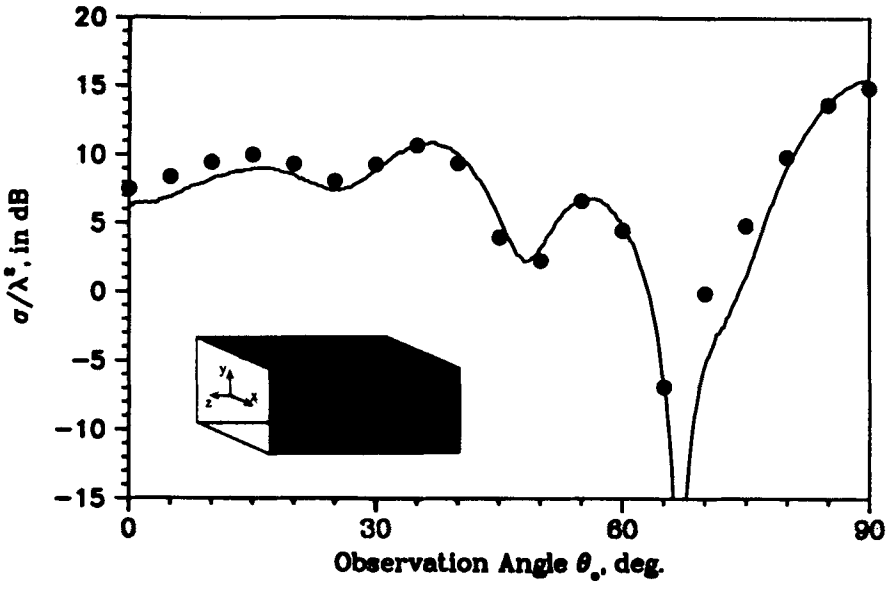



Figure 6. Backscatter pattern of a metallic rectangular inlet (1λ × 1λ × 1·5λ) for HH polarization. Black dots indicate computed values and the solid line represents measured data

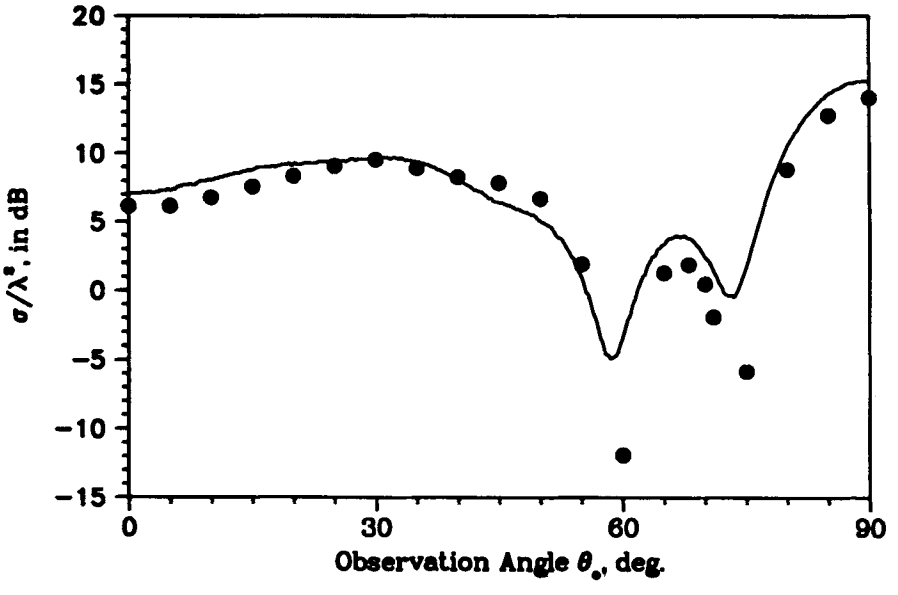



Figure 7. Backscatter pattern of a metallic rectangular inlet (1λ × 1λ × 1·5λ) for VV polarization. Black dots indicate computed values and the solid line represents measured data

## 6. APPENDIX

In this appendix, we present the algorithm for the modified ILU preconditioner. It is assumed that the data is stored in CSR format and that the column numbers for each row are sorted in increasing order. The sparse matrix is stored in the vector $X$ and the column numbers in $PC$. $SIG(i)$ contains the total number of non-zeros till the $i$th row. The locations of the diagonal entries for each row are stored in the vector $DIAG$. The preconditioner is stored in another complex vector, $LU$.

```
for i=1 step 1 until n-1 do
begin
  lbeg=diag(i)
  lend=sig(i)
  for j=lbeg+1 step 1 until lend do
  begin
      jj=pc(j)
    ij=srch(jj,i)
    if (ij.ne.0) then
    begin
          lu(ij)=lu(ij)/lu(lbeg)
    end
  end
end
```

ACKNOWLEDGEMENTS

This work was carried out on the facilities provided by NSF Grant CDA-92-14296 and the U-M Center for Parallel Computing and NASA-Ames Grants NAG 2-541 and NCA 2-653.

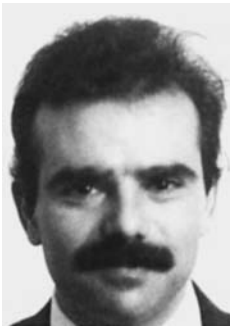
REFERENCES

1. A. Chatterjee, J. M. Jin and J. L. Volakis, 'Application of edge-based finite elements and ABCs to 3-D scattering', *IEEE Trans. Antennas Propagat.*, **AP-41**, 221–226 (1993).
2. D. S. Katz, M. J. Picket-May, A. Taflove and K. R. Umashankar, 'FDTD analysis of electromagnetic wave radiation from systems containing horn antennas', *IEEE Trans. Antennas Propagat.*, **AP-39**, 1203–1212 (1991).
3. J. P. Webb and V. N. Kanellopoulos, 'Absorbing boundary conditions for finite element solution of the vector wave equation', *Microwave and Opt. Tech. Letters*, **2**, 370–372 (1989).
4. M. L. Barton and Z. J. Cendes, 'New vector finite elements for three-dimensional magnetic field computation', *J. Appl. Phys.*, **61**, 3919–3921 (1987).
5. D. R. Kincaid and T. C. Oppe, 'ITPACK on supercomputers', *Numerical Methods, Lecture Notes in Mathematics*, Vol. 1005, Springer, Berlin, 1982, pp. 151–161.
6. G. V. Paolini and G. Radicati di Brozolo, 'Data structures to vectorize CG algorithms for general sparsity patterns', *BIT*, **29**, 703–718 (1989).
7. M. R. Hestenes and E. Stiefel, 'Methods of conjugate gradients for solving linear systems', *J. Res. Natl. Bur. Stand.*, **49**, 409–436 (1952).
8. P. Sonneveld, 'CGS, a fast solver for nonsymmetric linear systems', *SIAM J. Sci. Stat. Comput.*, **10**, 35–52 (1989).
9. R. Freund, 'Conjugate-gradient type methods for linear systems with complex symmetric coefficient matrices', *SIAM J. Sci. Stat. Comput.*, **13**, 425–448 (1992).
10. H. P. Langtangen, 'Conjugate gradient methods and ILU preconditioning of non-symmetric matrix systems with arbitrary sparsity patterns', *Int. J. Numer. Meth. Fluids*, **9**, 213–233 (1989).
11. J. R. Lovell, 'Hierarchical basis functions for 3D finite element methods', *ACES Digest*, 657–663 (1993).
12. E. Rothberg and A. Gupta, 'Parallel ICCG on a hierarchical memory multiprocessor—addressing the triangular solve bottleneck', *Parallel Computing* **18**, 719–741 (1992).
13. D. Windheiser, E. Boyd, E. Hao, S. G. Abraham and E. S. Davidson, 'KSR1 multiprocessor: analysis of latency hiding techniques in a sparse solver', *Proc. of the 7th International Parallel Processing Symposium*, Newport Beach, April 1993.
14. A. Woo, M. Schuh, M. Simon, T. G. Wang and M. L. Sanders, 'Radar cross-section measurement data of a simple rectangular cavity', Technical Report NWC TM7132, Naval Weapons Center, China Lake, CA, December 1991.

**Authors' biographies:**

**Arindjam Chatterjee** was born in Calcutta, India, in 1966. He received the B. Tech. (Honours) degree in electronics and electrical communication engineering from the Indian Institute of Technology, Kharagpur, in 1989 and the M.S. degree in electrical engineering from the University of Michigan, Ann Arbor, in 1991. Currently, he is working towards the Ph.D. degree at the Radiation Laboratory of the University of Michigan.

His research interests include partial differential equation methods like finite elements for electromagnetic field simulation in open and closed domain problems, sparse matrix techniques and parallel computing.



**John L. Volakis** was born in 1956 in Chios, Greece, where he also attended the Gymnasium of Males. He obtained his B.E. degree, *summa cum laude*, in 1978 from Youngstown State University, Youngstown, OH, the M.Sc. in 1979 from the Ohio State University, Columbus, Ohio, and the Ph.D. degree in 1982, also from the Ohio State University.

He has been with the University of Michigan, Ann Arbor, since 1984 where he is now a Professor in the Electrical Engineering and Computer Science (EECS) Department. From 1982–1984 he was with Rockwell International, Aircraft Division, and during 1978–1982 he was a Graduate Research Associate at the Ohio State University ElectroScience Laboratory. His primary research interests are in the development of analytical and numerical techniques as applied to electromagnetics. In 1993 he received the University of Michigan EECS Department Research Excellence Award.

Dr Volakis has served at various posts of the local IEEE AP/MTT/ED Southeastern Michigan Chapter from 1985 to 1988, as an Associate Editor of the *IEEE Transactions on Antennas and Propagation* from 1988 to 1992, and chaired the 1993 IEEE Antennas and Propagation Society Symposium and Radio Science Meeting. He is currently an associate editor for *Radio Science* and the *IEEE Antennas and Propagation Society Magazine*. He is a Senior member of the IEEE and a member of Sigma Xi, Tau Beta Pi, Phi Kappa Phi, and Commission B of URSI.

**Daniel Windheiser** graduated from the Ecole Polytechnique, France, in 1988 and received his doctoral degree from the University of Rennes I in May 1992. From July 1992 to July 1993, he was a Visiting Research Fellow with the Department of Electrical Engineering and Computer Science at the University of Michigan in Ann Arbor. Currently, he is in charge of High Performance Computing at DRET (Direction des Recherches et Etudes Techniques) in France and is a scientific adviser at SEH (Site Experimental en Hyperparallelisme). His research interests are in the area of parallel processing, compilers and software environments for massively parallel processors.