Scalable Solutions to Integral-Equation and Finite-Element Simulations

Tom Cwik, *Senior Member, IEEE*, Daniel S. Katz, *Member, IEEE*, and Jean Patterson

Invited Paper

***Abstract*— When developing numerical methods, or applying them to the simulation and design of engineering components, it inevitably becomes necessary to examine the scaling of the method with a problem's electrical size. The scaling results from the original mathematical development; for example, a dense system of equations in the solution of integral equations, as well as the specific numerical implementation. Scaling of the numerical implementation depends upon many factors; for example, direct or iterative methods for solution of the linear system, as well as the computer architecture used in the simulation. In this paper, scalability will be divided into two components—scalability of the numerical algorithm specifically on parallel computer systems and algorithm or sequential scalability. The sequential implementation and scaling is initially presented, with the parallel implementation following. This progression is meant to illustrate the differences in using current parallel platforms and sequential machines and the resulting savings. Time to solution (wall-clock time) for differing problem sizes are the key parameters plotted or tabulated. Sequential and parallel scalability of time harmonic surface integral equation forms and the finite-element solution to the partial differential equations are considered in detail.**

***Index Terms*—Finite-element methods, integral equations.**

I. Introduction

THE application of advanced computer architecture and software to a broad range of electromagnetic problems has allowed more accurate simulations of electrically larger and more complex components and systems than previously available. Computational algorithms are used in the design and analysis of antenna components and arrays, waveguide components, semiconductor devices, and in the prediction of radar scattering from aircraft, sea surface or vegetation, and atmospheric particles, among many other applications. A scattering algorithm, for example, may be used in conjunction with measured data to accurately reconstruct geophysical data. When used in design, the goal of a numerical simulation is to limit the number of trial fabrication and measurement iterations needed. The algorithms are used over a wide range of frequencies, materials and shapes, and can be developed to be specific to a single geometry and application, or more generally applied to a class of problems. The algorithms are numerical solutions to a mathematical model developed in the time or frequency domain and can be based on the integral equation or partial differential equation form of Maxwell's equations. A survey of the many forms of mathematical modeling and numerical solution can be found in [1], [2]. The goal of this paper is to examine the scalability of certain numerical solutions both generally as sequential algorithms, and then, specifically, as parallel algorithms using distributed memory computers. Parallel computers are evolving, surpassing traditional computer architectures by offering the largest memories and fastest computational rates to the user, and they will continue to evolve for several more generations in the near future [3].

This paper provides an overview of solutions to Maxwell's equations implicitly defined through systems of linear equations. Sequential and parallel scalability of time-harmonic surface integral-equation forms and the finite-element solution to the partial differential equations are considered. More general scalability of other sequential algorithms can be found in [4]. Initially, in Section II, a short review of the scalability of parallel computers is presented. In Sections III and IV, respectively, specific implementations of the MoM solution to integral-equation modeling and a finite-element solution will be discussed. The scalability of sequential solutions and related reduced memory methods will be considered, followed by an examination of parallel scalability and computer performance for these algorithms. The size of problems capable of being examined with current computer architectures will then be listed, with scalings to larger size problems also presented.

II. Scalability of Parallel Computers

Parallel algorithm scalability is examined differently from sequential algorithm scalability or, more precisely, scalability on shared memory (common address space) machines with no interprocessor communication overhead. Since a parallel computer is an ensemble of processors and memory linked by high-performance communication networks, calculations that were performed on a single processor must be broken into pieces and spread over all processors in use. At intermediate points in the calculation, data needed in the next stage of the calculation must be communicated to other processors.

The central consideration in using a parallel computer is to decompose the discretized problem among processors so

Manuscript received May 21, 1996; revised October 8, 1996. This work was supported by the Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration. D. S. Katz was supported by the Parallel Application Technology Program at Cray Research. The Cray Supercomputer used in this investigation was provided by funding from the NASA Offices of Mission to Planet Earth, Aeronautics, and Space Science.

The authors are with the Jet Propulsion Laboratory, California Institute of Technology, Pasadena, CA 91109 USA.

Publisher Item Identifier S 0018-926X(97)02303-X.

that the storage and computational load are balanced, and the amount of communication between processors is minimal [5], [6]. When this is not handled properly, efficiency is lower than 100%, where 100% is the machine performance when all processors are performing independent calculations and no time is used for communication. If the problem is decomposed poorly, some processors will work while others stand idle, thereby lowering machine efficiency. Similarly, if calculations are load balanced but processors must wait to communicate data, the efficiency is lowered. Scalability and efficiency are defined to quantify the parallel performance of a machine. Scalability (also termed speedup) is the ratio of time to complete calculations sequentially on a single processor to that on P processors

$$S = \frac{T_{seq}}{T(P)}. \tag{1}$$

The efficiency is then the ratio of scalability to the number of processors

$$\varepsilon = \frac{S}{P}. \tag{2}$$

If an algorithm issues no communication calls, and there is no component of the calculation that is sequential and, therefore, redundantly repeated at each processor, the scalability is equal to the number of processors P and the efficiency is 100%. The scalability, as defined, must be further clarified if it is to be meaningful since the amount of storage, i.e., problem size, has not been included in the definition. Two regimes can be considered—fixed problem size and fixed grain size. The first, fixed problem size, refers to a problem that is small enough to fit into one or a few processors and is successively spread over a larger sized machine. The amount of data and calculation in each processor will decrease and the amount of communication will increase. The efficiency must, therefore, successively decrease, reaching a point where CPU time is communication bound. The second, fixed grain size problems, refers to a problem size that is scaled to fill all the memory of the machine in use. The amount of data and calculation in each processor will be constant, and in general, much greater than the required amount of communication. Efficiency will remain high as successively larger problems are solved. Fixed grain problems ideally exhibit scalability that is a key motivation for parallel processing; successively larger problems can be mapped onto successively larger machines without a loss of efficiency.

When developing numerical methods for electromagnetics, or applying them to the simulation and design of engineering components, it inevitably becomes necessary to examine the scaling of the method with a problem's electrical size. The scaling results from the original mathematical development; for example, a dense system of equations in the solution of integral equations, as well as the specific numerical implementation. Scaling of the numerical implementation depends upon many factors; for example, direct or iterative methods for solution of the linear system, as well as the computer architecture used in the simulation. In the rest of this paper, scalability will be divided into two components—algorithmic scalability and scalability of the numerical algorithm specifically on parallel computer systems. Algorithmic scalability refers to the amount of computer memory and time needed to complete an accurate solution as a function of the electrical size of the problem. Scalability on a parallel computer system refers to the ability of an algorithm to achieve performance proportional to the number of processors being used as outlined above. The sequential implementation and scaling is initially presented with the parallel implementation following. This progression is meant to illustrate the differences in using current parallel platforms versus sequential machines, and the resulting savings. Clearly, different mathematical formulations can lead to alternate numerical implementations and different scalings. The objective of numerical modeling is to provide an accurate simulation of the measurable quantities in an amount of time that is useful for engineering design. With this objective, time to solution (wall-clock time) for differing problem sizes is the key parameter plotted or tabulated.

III. Integral Equation Formulations

The method of moments (MoM) is a traditional algorithm used for the solution of a surface integral equation [7], [8]. This technique can be applied to impenetrable and homogenous objects, objects where an impedance boundary condition accurately models inhomogenous materials or coatings, and those inhomogenous objects that allow the use of Green's functions specific to the geometry. A dense system of equations results from discretizing the surface basis functions in a piecewise continuous set, with this system being solved in various ways. The components of MoM solutions that affect the algorithmic scalability are the matrix fill, and matrix solution. A system of equations

$$\mathbf{AX} = \mathbf{B} \tag{3}$$

results from the MoM, where $\mathbf{A}$ generally is a nonsymmetric complex-valued square matrix, and $\mathbf{B}$ and $\mathbf{X}$ are complex-valued rectangular matrices when multiple excitations and solution vectors are present. The solution of (3) is most conveniently found by an LU factorization

$$\mathbf{A} = \mathbf{LU} \tag{4}$$

where $\mathbf{L}$ and $\mathbf{U}$ are lower and upper triangular factors of $\mathbf{A}$. The solution for $\mathbf{X}$ is computed by successive forward and back substitutions to solve the triangular systems

$$\begin{aligned} \mathbf{LY} &= \mathbf{B} \\ \mathbf{UX} &= \mathbf{Y}. \end{aligned} \tag{5}$$

Because the system in (3) is not generally positive definite, rows of $\mathbf{A}$ are permuted in the factorization, leading to a stable algorithm [9]. Table I is a listing of computer storage and time scalings for a range of problem sizes when using standard LU decomposition factorization and readily available forward and backward solution algorithms [10]. The table is divided along columns into problem size, factorization, and solution components. The first column fixes the memory size of the machine being used; the number of unknowns and surface area modeled (assuming 200 unknowns/λ^2) based on this memory size are

TABLE I
SCALING OF METHOD OF MOMENTS MATRIX FACTORIZATION AND SOLUTION ALGORITHMS

PROBLEM SIZE			FACTOR			SOLVE (1 excitation)		
MEMORY (Mbytes)	N	AREA (λ^2)	0.1 Gflops machine performance	10	1000	0.1 Gflops machine performance	10	1000
128	2,800	14	12.2 mins	0.12 mins	0.001 mins	1.25 secs	10^{-2} secs	10^{-4} secs
256	4,000	20	35.6	0.36	0.004	2.55	2.5×10^{-2}	2.5×10^{-4}
512	5,600	28	97.6	0.98	0.010	5.01	5.0×10^{-2}	5.0×10^{-4}
1024	8,000	40	284.4	2.84	0.028	10.21	0.10	0.001
8192	22,627	113	107.3 hrs	1.07 hrs	0.017 hrs	1.36 mins	10^{-2} mins	10^{-4} mins
32,768	45,254	226	-	8.58	0.086	-	5.5×10^{-2}	5.5×10^{-4}
131,072	90,509	452	-	68.65	0.687	-	0.22	0.002
524,288	181,019	905	-	-	5.492	-	-	0.009

in the next columns. The factor and solve times are based on the performance of three classes of machine, current high-end workstation (0.1 Gigaflops), current supercomputer (10 Gigaflops), and next generation computer (1000 Gigaflops). Similarly, the rows are divided into the top four which would typically correspond to current generation workstations, and the lower four that correspond to supercomputer class machines. Due to the nature of the dense matrix data structures [10] in the factorization algorithms, this component of the calculation can be highly efficient on general computers. A value of 80% efficiency of the peak machine performance is used for the time scalings. The backward and forward solution algorithms though operate sequentially on triangular matrix systems, resulting in reduced performance, and a 50% efficiency is used in these columns. This performance will increase when many excitations (right-hand sides) are involved in the calculation, resulting in performance closer to peak.

The time for factorization, scaling as N^3, can limit the time of the calculation for large problems, even if the matrix can be assembled and stored. An alternative to direct factorization methods is the use of iterative solvers of the dense system [11], [12]. These methods are even more useful if multiple right-hand sides are present, since the iterative solvers exploit information in these vectors [11]. If the convergence rate can be controlled and the number of iterations required to complete a solution are limited, the solution time can be reduced as compared to that of the direct factorization methods.

The direct limitation of integral-equation methods is the memory needed to store the dense matrix. Larger problems can only be solved by circumventing this bottleneck. One approach to this problem is to use higher order parametric basis functions which reduce the number of unknowns needed to model sections of the surface [13]. However, it is clear that the growth in memory required for storage of the dense complex impedance matrix (N^2), cannot be overcome by only slightly reducing the size of N or by increasing the number of computer processors applied to the problem. Alternative methods are, thus, desirable.

One such method uses special types of basis functions to produce a sparse impedance matrix [14], reducing the storage from the N^2 to αN where α is a constant independent of N. The resulting sparse system can be solved using the methods described in latter sections of this paper. Other classes of methods for generating and solving the impedance matrix while reducing storage are summarized in [15], while another is the fast multipole method [16], [17]. This approach decomposes the impedance matrix in a manner that also reduces the needed storage to αN. The fast multipole method has been parallelized in [18], where it is suggested that an Intel Paragon system with 512 nodes each containing 32 Mbytes of memory could solve a problem with 250 000 basis functions.

A. Scalability on Parallel Computers

To specifically examine parallel scalability, the electric-field integral-equation model developed in the PATCH code [19], [20] is used. This code uses a triangularly faceted surface model of the object being analyzed, and builds a complex dense matrix system [identical to that given in (3)]

$$\mathbf{Z}\,\mathbf{I} = \mathbf{V} \tag{6}$$

where

$$\mathbf{Z}_{i,j} = \oiint_{\partial S} \mathbf{T}_i(\overline{r})\, d\overline{r} \oiint_{\partial S} \mathbf{J}_j(\overline{r'})\, \mathrm{G}(\overline{r} - \overline{r'})\, d\overline{r'} \tag{7}$$

and $\mathbf{i}$ and $\mathbf{j}$ are indices on the edges of the surface facets, G is the Green's function for an unbounded homogeneous space, $\overline{r}$ and $\overline{r'}$ are arbitrary source and observation points, and $\{\mathbf{T}_i, \mathbf{J}_j\}$ are current testing and expansion functions. The impedance matrix ($\mathbf{Z}$) is factored by means of an LU factorization, and for each vector $\mathbf{V}$ a forward and backward substitution is performed to obtain $\mathbf{I}$, the unknown currents on the edges of the surface facets. It is then a simple matter to compute radar cross section (RCS) or other field quantities by a forward integral. The elements of the PATCH code considered in the parallelization are matrix fill, matrix factorization and solution of one or many right-hand sides, and the calculation of field quantities. The computation cost of these three elements must be examined in relation to increasing problem size and increasing number of processors to understand the scalability of the PATCH code.

Matrix Equation Fill: Since the impedance matrix is a complex dense matrix of size N, where N is the number of edges used in the faceted surface of the object being modeled, the matrix has N^2 elements, and filling this matrix scales as N^2. One method for reducing the amount of time spent in this operation when using a parallel computer is to spread the fixed number of elements to be computed over a large number of processors. Since the amount of computation is theoretically fixed (neglecting communication between

processors), applying P processors to this task should provide a time reduction of P. However, the computations required by the PATCH code's basis functions involve calculations performed at the center of each patch that contribute to the matrix elements associated with the three current functions on the edges of that patch. The sequential PATCH code will loop over the surface patches and compute partial integrals for each of the three edges that make up that patch. This algorithm is quite efficient in a sequential code, but is not as appropriate for a parallel code where the three edges of both the source and testing patch (corresponding to the row and column indexes of the matrix) will not generally be located in the same processor. The parallel version of PATCH currently only uses this calculation for those matrix elements that are local to the processor doing the computation; introducing an inefficiency compared to a sequential algorithm. This inefficiency is specific to the integration algorithm used and can be removed, for example, by communicating partial results computed in one processor to the processors that need them. This step has not been taken at this time in the parallel PATCH code partially due to the complexity it adds to the code.

Matrix Equation Factorization and Solution: When the impedance matrix is assembled, the solution is completed by a LU factorization, scaling as N^3, and a forward and backward substitution used to solve for a given right-hand side, scaling as N^2. The choice of the LU factorization algorithm determines the matrix decomposition scheme used on the set of processors.

One style of decomposition that is suitable for use with LINPACK [21] factorization routines is to partition the matrix

$$\mathbf{A} = \begin{pmatrix} \mathbf{A}_0 \\ \hline \vdots \\ \hline \mathbf{A}_{(p_n-1)} \end{pmatrix} \tag{8}$$

where $\mathbf{A}_i \in C^{m_i \times N}$ and $m_i \approx N/P$. Each submatrix $\mathbf{A}_i$ is then assigned to processor i, where P is the number of processors. The LINPACK method involves many BLAS 1 (vector–vector) operations [22], which perform at 25–70 MFLOPS on the Cray T3D (150 MFLOPS peak performance.) These type operations perform poorly on the T3D and other hierarchical memory computers because the amount of work that is performed on the data brought from memory to the cache then to the processor is similar in size to the amount of that data.

Another type of decomposition is to assume that the physical processors, $\mathbf{P}_{ij}$ form a logical two-dimensional (2-D) $p_r \times p_c$ array, where i and j refer to the row and column index of the processor. Then the simplest assignment of matrix elements to processors in a block method is to partition the matrix

$$\mathbf{A} = \begin{pmatrix} \mathbf{A}_{00} & \cdots & \mathbf{A}_{0(p_c-1)} \\ \hline \vdots & \ddots & \vdots \\ \hline \mathbf{A}_{(p_r-1)0} & \cdots & \mathbf{A}_{(p_r-1)(p_c-1)} \end{pmatrix} \tag{9}$$

where $\mathbf{A}_{ij} \in C^{m_i \times n_j}$, $m_i \approx N/p_r$, and $n_j \approx N/p_c$. Submatrix $\mathbf{A}_{ij}$ is then assigned to processor $\mathbf{P}_{ij}$. This is an appropriate decomposition for use with LAPACK routines

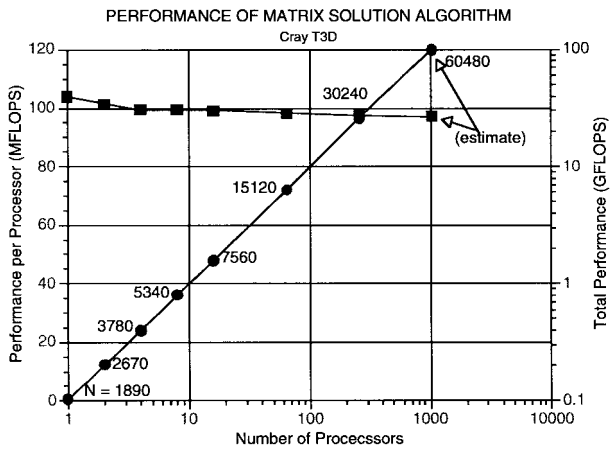


Fig. 1. Performance for BSOLVE (includes factoring matrix, estimating condition number, and solving for one right-hand side).

[10] that use BLAS 3 (matrix–matrix) operations [23] and can perform at a high rate, typically 100–120 MFLOPS on a T3D processor. These operations perform a large amount of work on the data that has been brought into the cache, compared with BLAS 1 operations. However, this simple decomposition doesn't provide good load balance. It can be overcome by blocking the matrix into much smaller $k \times k$ submatrices, and *wrapping* these in two dimensions onto the logical processor array. In other words, partition

$$\mathbf{A} = \begin{pmatrix} \mathbf{A}_{00} & \cdots & \mathbf{A}_{0(M-1)} \\ \hline \vdots & \ddots & \vdots \\ \hline \mathbf{A}_{(M-1)0} & \cdots & \mathbf{A}_{(M-1)(M-1)} \end{pmatrix} \tag{10}$$

where $M \approx N/k$, and all blocks are of size $k \times k$. Then, block $\mathbf{A}_{ij}$ is assigned to processor $\mathbf{P}_{(i \bmod p_r)(j \bmod p_c)}$. This is the partitioning strategy that is used by PATCH. (On the T3D, $k = 32$ has been found to provide optimal results between load balance demanding the smallest possible k and the performance of the BLAS 3 operations requiring a large value for k.)

The PATCH code uses a matrix equation solver (named BSOLVE [24]) based on this decomposition. Fig. 1 shows the total and per processor performance for BSOLVE for the largest matrix that can be solved on each size of machine. Total time scales at the same efficiency as total performance, and is 25 min for the 30 240-size matrix on 256 processors.

Each processor of the T3D (60 Mbytes usable memory) can hold in memory a matrix piece of size 1890 $\times$ 1890. The largest T3D has 1024 processors and can store a matrix of size 60 480 $\times$ 60 480 and can be factored in about 100 min. The matrix solution algorithm is a member of a class of problems (those that are limited only by memory requirements and not by run-time requirements) that are good candidates for out-of-core methods. This would require storing a larger matrix equation on disk, and loading portions of the matrix into memory for the factorization and solution steps. For an out-of-core solution to be efficient, the work involved in loading part of a problem from disk must be overlapped with the work performed in solving the problem, so that storing the matrix on disk doesn't significantly increase the run time over what it

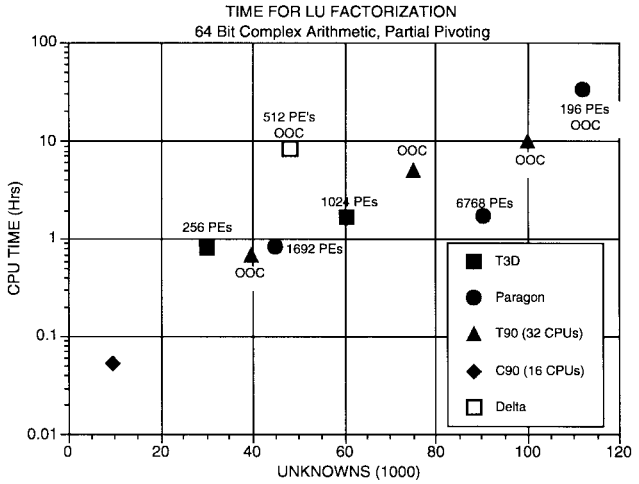


Fig. 2. CPU time versus number of unknowns for 64-b complex dense direct solvers with partial pivoting for various machines (Cray T3D, T90, and C90, Intel Paragon, and Delta). Processing elements (PE's) used in calculation shown for parallel computers. In-core and out-of-core (OOC) results specified.

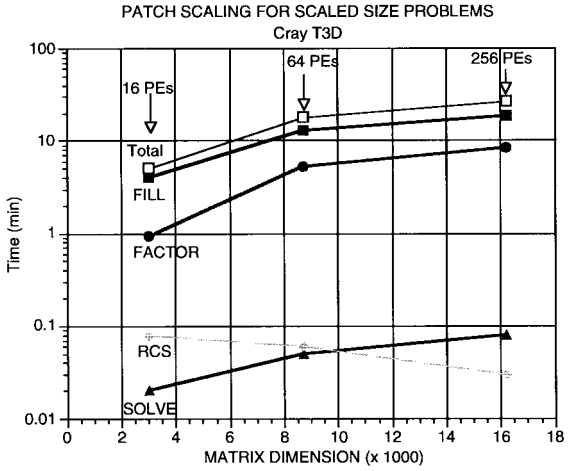


Fig. 3. Scalability for PATCH code for scaled size problems.

would be on a machine with more memory. Specialized matrix equation solvers have been developed to efficiently exploit the large disk memory available on many parallel machines. Fig. 2 shows general results for dense solvers, both in core and out of core, for various machines.

Computation of Observables: The observables of the PATCH code, such as RCS information or near- or far-field quantities, can be easily computed once the currents on the edges of the surface patches are known. These calculations involve forward integrals of the currents and the freespace Green's function. Because this current is discretized, the integration results in a summation of field components due to each current basis function. Since these discretized currents are distributed over all the processors in a parallel simulation, partial sums can be performed on the individual processors, followed by a global summation of these partial results to find the total sum. This calculation scales quite well since the only overhead is the single global sum.

Total Performance: Fig. 3 shows overall code timings for a scaled size problem. Each case involves a matrix that fills the same fraction of each processor's memory. As the problem

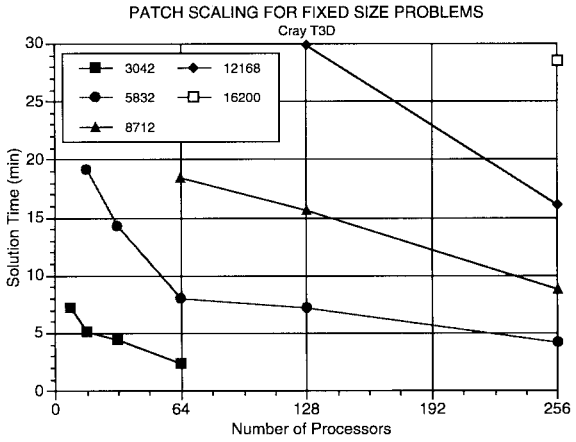


Fig. 4. Scalability for PATCH code for fixed size problems. The 16 200 unknown case was only run on 256 processors.

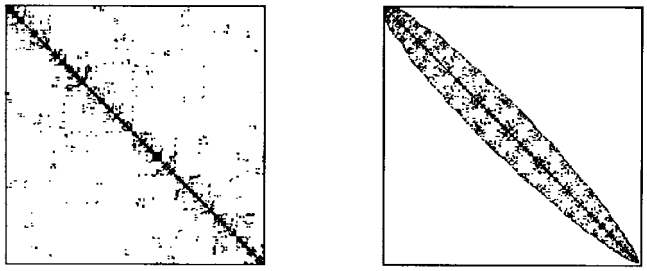
size increases, more processors are used in the calculation. It is clear that the matrix fill is dominant in the code, and in fact, the crossover point between the $O(n^3)$ factorization and the $O(n^2)$ fill has not been reached. Also, the time involved in the radar cross section calculation is shown to decrease as the number of processors is increased, as was described in the previous paragraph. It may also be observed that the matrix solve time, also $O(n^2)$, parallels the fill time quite well.

Fig. 4 shows scalings for the fixed size problems on the T3D. Each of the problems shown is run on the smallest set of processors needed to hold the matrix data, and then on larger numbers. The total code time initially decreases linearly with the number of processors, leveling off as the amount of communication time begins to become a larger fraction of the total time needed to complete the calculations.

IV. Finite-Element Formulations

Volumetric modeling by the use of an integral equation can also be used in simulations, though the available memory of current or planned technology greatly limits the size of problems that can be modeled. Because of this limitation in modeling 3-D space by integral equations, finite-element solutions of the partial differential equations that lead to sparse systems of equations are commonly used [25]. A finite-element model is natural when the problem contains inhomogenous material regions that surface integral equation methods are either incapable of modeling or are very costly to model. The problem domain is broken into a finite-element basis function set used to discretize the fields. The resulting linear system of equations—rather than scaling as the N^2 storage of the MoM—scales as mN where m is the average number of nonzero matrix equation elements per row of the sparse linear system. This value is dependent upon the order of the finite element used, but is typically between 10 and 100, and is independent of the size of the mesh. For a six unknown, vector edge-based tetrahedral finite element [26], m is typically 16.

Typically, the system of equations resulting from a finite element discretization is symmetric; the nonzero structure of a representative example is shown in Fig. 5(a). A symmetric fac-



(a) (b)

Fig. 5. Nonzero matrix structure of typical finite-element simulation. (a) Original structure. (b) Structure after reordering to minimize bandwidth.

torization of this system (Cholesky factorization) leads to [27]

$$\begin{aligned}\mathbf{A} &= \overline{\mathbf{L}}\mathbf{D}\overline{\mathbf{L}}^{\mathbf{T}} \\ &= \overline{\mathbf{L}}\mathbf{D}^{\mathbf{1/2}}\mathbf{D}^{\mathbf{1/2}}\overline{\mathbf{L}}^{\mathbf{T}} \\ &= \mathbf{L}\mathbf{L}^{\mathbf{T}}\end{aligned} \tag{11}$$

where the diagonal matrix is specifically shown distributed symmetrically between the symmetric factors $\mathbf{L}$, an important consideration when symmetrically applying an incomplete Cholesky preconditioner in iterative methods. The factorization results in nonzero elements in $\mathbf{L}$ where nonzeros exist in $\mathbf{A}$, as well as fill-in, or new nonzero entries generated during the factorization. Fill-in requires additional storage for $\mathbf{L}$, as well as additional time to complete the factorization. To reduce the amount of fill-in, the system is reordered by applying a permutation

$$(\mathbf{PAP}^{\mathbf{T}})\mathbf{PX} = \mathbf{PB} \tag{12}$$

where the permutation matrix $\mathbf{P}$ satisfies $\mathbf{PP}^{\mathbf{T}} = \mathbf{I}$. In (12), $\mathbf{PAP}^{\mathbf{T}}$ remains symmetric, and the forward and backward substitution phases become

$$\begin{aligned}\hat{\mathbf{L}}\mathbf{Y} &= \mathbf{PB} \\ \hat{\mathbf{L}}^{\mathbf{T}}\mathbf{Z} &= \mathbf{Y} \\ \mathbf{X} &= \mathbf{P}^{\mathbf{T}}\mathbf{Z}\end{aligned} \tag{13}$$

where $\hat{\mathbf{L}}$ is the Cholesky factor of $\mathbf{PAP}^{\mathbf{T}}$. For a sparse factorization, the permutation matrix is chosen to minimize the amount of fill-in generated. Since there are $n!$ possibilities for $\mathbf{P}$ to minimize fill-in, heuristic methods are used to achieve a practical minimization, the most common being the minimum degree algorithm [28]. Fig. 5(b) shows the nonzero structure of the representative system after reordering for a canonical scattering problem.

Table II lists scaling data for problem size when using a Cholesky factorization with the minimum degree reordering algorithm used to minimize storage. Based on the computer storage available, the number of edges in a edge-based tetrahedral mesh [18] along with the number of nonzeros in the factor $\hat{\mathbf{L}}$ [29], and the volume that can be modeled is shown. The volume is based on the use of 15 000 tetrahedra per cubic wavelength, corresponding to approximately 25 edges per linear wavelength. This number can vary depending on the physical geometry (curvature, edges, points) and the local nature of the fields.

TABLE II
SCALING OF TYPICAL FACTORIZATION FINITE-ELEMENT MATRIX SOLUTION ALGORITHMS

PROBLEM SIZE			
MEMORY (Mbytes)	N 10^3	$\lvert\hat{\mathbf{L}}\rvert$ 10^6	VOL (λ^3)
128	30	8	2.0
256	50	16	3.3
512	82	32	5.5
1024	135	64	9.0
8,192	600	512	40
32,768	1,627	2,048	108
131,072	4,410	8,192	294

Theoretically, the system in (12) is not generally positive definite, being symmetric indefinite, and a Cholesky factorization in this case is numerically unstable. For the practical solution of most problems, this has not been found to be a issue. Methods that preserve the symmetric sparsity, and allow pivoting to produce more stable algorithms can be used [28].

From Table II it is seen that even though the storage for the finite-element method is linear in N, the fill-in due to the use of factorization algorithms causes the storage to grow as $N^{1.4}$. Linear storage can be maintained using an iterative solution. Sparse iterative algorithms for systems resulting from electromagnetic simulations recursively construct Krylov subspace basis vectors that are used to iteratively improve the solution to the linear system. The iterates are found from minimizing a norm of the residual

$$\mathbf{r} = \mathbf{Ax} - \mathbf{b} \tag{14}$$

at each step of the algorithm. (A single solution vector for a single excitation is shown.) Since the system is complex-valued indefinite, methods appropriate for this class of system such as bi-conjugate gradient, generalized minimal residual, and the quasi-minimal residual algorithm are applied [30]. They all require a matrix–vector multiply, and a set of vector inner products for the calculation. The iterative algorithms require the storage of the matrix and a few vectors of length N. When only the matrix and a few vectors need to be stored, problems of very large size can be handled, if the convergence rate is controlled. The number of iterations (with a sparse matrix-dense vector multiply accounting for over 90% of the time at each step of the iterative algorithm) determines the time to solution.

Because the matrix–vector multiply dominates the Krylov iterative methods, the algorithmic scaling is found from this operation. A single sparse matrix-dense vector multiply requires mN operations and, if there are I total iterations required for convergence, the number of floating point operations needed is $I{*}mN$. A typical solution of the system of equations, without the application of a preconditioner, may require a number of iterations $I = \sqrt{N}$, producing the sequential algorithm scaling

$$mN^{3/2} \tag{15}$$

for the solution of a single right-hand side. The direct factorization algorithm scales as m^2N [28, p. 104]; therefore, the factorization methods—when memory is sufficient to hold the fill-in entries—give considerable central processing unit (CPU) time savings in the solution. Further advantage is also gained over iterative methods when using a direct factorization. Modern computer architectures are typically much less efficient at performing the sparse operations required in the sparse matrix-dense vector multiply of the iterative algorithm, as compared to operations on dense matrix systems. Current direct factorization methods attempt to use block algorithms, exploiting dense matrix substructure in the sparse system and, therefore, increasing the performance of the factorization, further improving the performance from that given by the algorithmic scaling differences.

It is seen that the number of iterations directly increases the time to solution in the iterative methods. This number can be controlled to some degree by the use of preconditioning methods that attempt to transform the matrix equation into one with more favorable properties for an iterative solution. To control the convergence rate, the matrix $\mathbf{A}$ should be scaled by a diagonal matrix that produces *ones* along the diagonal. This scaling removes the dependence on different element sizes in a mesh. A preconditioner $\mathbf{M}$ can then be symmetrically applied to transform the system giving

$$\mathbf{M^{-1}AM^{-1}(Mx)} = \mathbf{M^{-1}b}. \tag{16}$$

The right-hand side vector is initially transformed, and the system is then solved for the intermediate vector $\overline{\mathbf{x}} = \mathbf{Mx}$, multiplying this vector by $\mathbf{M^{-1}}$, $\mathbf{A}$, and $\mathbf{M^{-1}}$ in succession at each iterative step. When the solution has converged, $\mathbf{x}$ is recovered from $\overline{\mathbf{x}}$. The closer $\mathbf{M}$ is to $\mathbf{L}$ in (11), the quicker the transformed system will converge to a solution. A common preconditioner is an incomplete Cholesky factorization [31] where $\mathbf{M}$ is chosen as a piece of the factor $\mathbf{L}$ in (11). It is computed to keep some fraction of the true factorization elements, with the exact number and sparsity location of the elements dependent on the exact algorithm used. A useful form of incomplete factorization keeps the same number of elements in the incomplete factor as there are in $\mathbf{A}$. This requires three times the number of operations at each iterative step; therefore the time to solution will be decreased if the number of iterations is lessened by two-thirds when applying this preconditioner.

When the right-hand side consists of a number of vectors, newly developed block methods can be applied to the system to use the additional right-hand sides to improve the convergence rate [32], [33].

A. Scalability on Parallel Computers

In a finite-element algorithm, the resultant sparse system of equations is stored within a data structure that holds only the nonzero entries of the sparse system. This sparse system must ultimately be distributed over the parallel computer, requiring special algorithms to either break the original finite element mesh up into specially formed contiguous pieces, or by distributing up the matrix entries themselves onto the processors of the computer. As in the dense MoM solution, the pieces are distributed in a manner that allows for an efficient solution of the matrix equation system.

The Finite-Element Mesh and the Sparse Matrix Equation: The volumetric region (V) is enclosed by a surface (∂V), in which a finite-element discretization of a weak form of the wave equation is used to model the geometry and fields

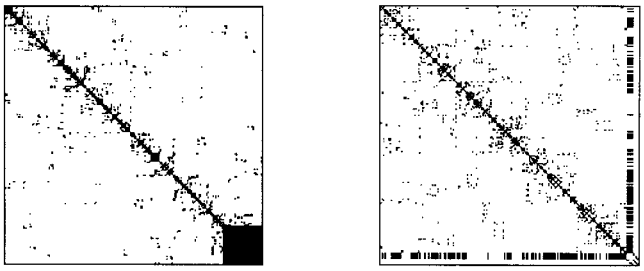
$$\frac{\eta_0}{jk_0}\iiint_V \left[\frac{1}{\varepsilon_r}(\nabla\times\overline{H})\bullet(\nabla\times\overline{W}^*) - k^2\mu_r\overline{H}\bullet\overline{W}^*\right]dv - \iint_{\partial V} \overline{E}\times\hat{n}\bullet\overline{W}^*\,ds = 0. \tag{17}$$

$\overline{H}$ is the magnetic field (the $\overline{H}$ equation is used in this paper; a dual $\overline{E}$ equation can also be written), $\overline{W}$ is a testing function, the asterisk denotes conjugation, and $\overline{E}\times\hat{n}$ is the tangential component of $\overline{E}$ on the bounding surface. In (17), ε_r and μ_r are the relative permittivity and permeability, respectively, and k_0 and η_0 are free-space wave number and impedance, respectively. A set of finite-element basis functions, the tetrahedral, vector-edge elements (Whitney elements) will be used to discretize (17),

$$\overline{W}_{mn}(r) = \lambda_m(r)\nabla\lambda_n(r) - \lambda_n(r)\nabla\lambda_m(r) \tag{18}$$

where $\lambda(r)$ are the tetrahedral shape functions and indexes (m, n) refer to the two nodal points of each edge of the finite-element mesh. These elements will be used for both expansion and testing (Galerkin's method) in the finite-element domain. Because of the local nature of (17), (subdomain basis functions and no Green's function involved in the integration of the fields), the system of equations resulting from the integration only contains nonzero entries when the finite elements overlap or are contiguous at an edge. Because the mesh is unstructured, containing elements of different size and orientation conforming to the geometry, the resultant matrix equation will have a sparsity structure that is also unstructured.

The sparsity structure is further altered by the form of the Sommerfeld boundary condition applied on the surface S. When local, symmetric absorbing conditions are applied on the boundary [34]—entering into the calculation through the surface integral in (17)—a matrix with the structure, shown in Fig. 5(a), results. It is seen that the diagonal is entirely filled, corresponding to the self terms in the volume integral in (17), with the m nonzero entries scattered along the row (or column) of the symmetric matrix. The location of these entries is completely dependent upon the ordering of the edges of the tetrahedral elements used in the discretization. If a different shape or order of the elements are used, the nonzero structure will differ slightly from the one shown. When an integral equation method is used to truncate the mesh [35], [36], a dense block of elements will appear in the lower right of the system (when the edges of the finite-element mesh on the boundary are ordered last), as shown in Fig. 6(a). The integral equation approach to truncating the mesh uses the finite-element facets on the boundary as source fields in an integral equation, resulting in a formulation for this piece of the calculation similar to that in Section III, and with an amount



(a) (b)

Fig. 6. (a) Nonzero matrix sparsity structure for system with dense surface integral-equation boundary condition applied. (b) Surface of revolution integral equation boundary condition. The mesh has 5343 edges, with 936 of those on the boundary.

of storage needed for the dense matrix as a function of the electrical surface area tabulated in Section II. To circumvent the large dense storage needed with this application of a global boundary condition, a surface of revolution can be used to truncate the mesh [37], [38], using a set of global basis functions to discretize the integral equation on this surface. This results in a system similar to that in Fig. 6(b) containing very small diagonal blocks due to the orthogonal global basis functions along the surface of revolution truncating the mesh, as well as a matrix coupling the basis functions in the integral-equation solution to the finite-element basis function on the surface. These coupling terms lead to banded thin rectangular matrices symmetric about the diagonal of the matrix. Other forms of the integral equation solution, as outlined in Section III, can also be used to discretize the integral equation modeling fields on the mesh boundary, leading to slight variations of the matrix systems shown in Fig. 6.

The systems graphically represented in Fig. 6 generally have the form

$$\begin{bmatrix} \mathbf{K} & \mathbf{C} \\ \mathbf{C}^{\dagger} & \mathbf{Z} \end{bmatrix} \begin{bmatrix} \mathbf{H} \\ \mathbf{I} \end{bmatrix} = \begin{bmatrix} \mathbf{0} \\ \mathbf{V} \end{bmatrix} \tag{19}$$

where $\mathbf{K}$ is the sparse, symmetric finite-element matrix, found from the volume integral in (17), $\mathbf{C}$ can be termed the coupling matrix that represents interactions between the finite elements at the boundary and the integral equation basis functions, and $\mathbf{Z}$ represents the integral equation, MoM matrix entries. The symbol † indicates the adjoint of a matrix. $\mathbf{H}$ is the vector of magnetic field coefficients for each finite element, and $\mathbf{I}$ represents the equivalent current basis functions on the boundary of the mesh. For a scattering problem formulation, the incident field couples only to the integral equation boundary, and is represented as $\mathbf{V}$. For radiation problems the $\mathbf{0}$ and $\mathbf{V}$ vectors are interchanged since the impressed source is modeled in the mesh. Differing formulations lead to variations in (19), but the general algebraic nature is preserved. To exploit the sparsity of $\mathbf{K}$ in (19), the system is solved in two steps by initially substituting $\mathbf{H} = -\mathbf{K}^{-1}\mathbf{C}\mathbf{I}$ from the first equation in (19) into the second, producing

$$(\mathbf{Z} - \mathbf{C}^{\dagger}\mathbf{K}^{-1}\mathbf{C})\mathbf{I} = \mathbf{V}. \tag{20}$$

This system's size is on the order of the number of basis functions in the integral-equation model, is dense, and can be solved by either direct factorization or iterative means as outlined in Section III. The intermediate calculation $\mathbf{K}\mathbf{X} = \mathbf{C}$ is the sparse system of equations to be solved, producing $\mathbf{X}$.

The solution of this sparse system on a parallel computer requires it to be distributed. Traditionally, the dependence between mesh data and the resultant sparse matrix data is exploited in the development of mesh-partitioning algorithms [39]–[42], [55]. These algorithms break the physical mesh or its graph into contiguous pieces that are then read into each processor of a distributed memory machine. The mesh is decomposed such that the pieces have roughly the same number of finite elements, and to some measure, each piece has minimal surface area. Since the matrix assembly routine [the volume integral in (17)] generates nonzero matrix entries that correspond to the direct interconnection of finite elements, the mesh partitioning algorithm attempts to create a load balance of the sparse system of equations. Processor communications in the algorithm that solves the sparse system are limited by the minimization of the surface area of each mesh piece.

Mesh-partitioning algorithms are generally divided into multilevel spectral partitioning, geometric partitioning, and multilevel graph partitioning. Spectral partitioning methods [40], [41] create eigenvectors associated with the sparse matrix, and use this information to recursively break the mesh into roughly equal pieces. They require the mesh connectivity information as input, and return lists of finite elements for each processor. Geometric partitioning [39] is an intuitive procedure that divides the finite-element mesh into pieces based on the geometric (node x, y, z coordinates) of the finite-element mesh. This algorithm requires the mesh connectivity as well as the node spatial coordinates and returns lists of finite elements for each processor. Graph partitioning [42] operates on the graph of the finite-element mesh (mesh connectivity information) to collapse (or coarsen) vertices and edges into a smaller graph. This smaller graph is partitioned into pieces, and then uncoarsened and refined for the final partitions of finite elements for the parallel processors. The input and output is identical to spectral methods. Multilevel algorithms operate by performing multiple stages of the partitioning simultaneously, accelerating the algorithm. Most of these algorithms and their offshoots perform similarly in practice, with the spectral and graph partitioning algorithms being simpler to use since they do not need geometry information. An alternative to these mesh-partitioning algorithms is a method that divides the matrix entries directly, without operating on the finite-element mesh. This will be examined in Section IV-A.

Different decompositions are used depending on whether direct factorization or iterative methods are used in the solution. Decompositions for iterative solutions, as well as the iterative methods themselves have shown greater ease in parallelization than direct factorization methods. Both approaches will now be considered.

Direct Sparse Factorization Methods: Direct factorization methods require a sequence of four steps; reordering of the sparse system to minimize fill-in, a symbolic factorization stage to determine the structure and storage of $\hat{\mathbf{L}}$ in (13), the numeric factorization producing the complex-valued entries

of $\hat{\mathbf{L}}$, and the triangular forward and backward solutions. The fundamental difficulty in the parallel sparse factorization is the development of an efficient reordering algorithm that both minimizes the fill-in and scales well on distributed memory machines while controlling the amount of communication necessary in the computation. The minimum degree algorithm typically used in sequential packages is inherently nonparallel, proceeding sequentially in the elimination of nodes in the graph representing the nonzero structure of the matrix. Other algorithms for reordering, as well as the following symbolic and numeric factorization steps that depend on this ordering are under study [43]. Current factorization algorithms [44]–[46] can exhibit fast parallel solution times on moderately large sized problems, but are dependent on the relative structure of the mesh, whether or not the problem is 2-D or 3-D, and the relative sparsity of the nonzero entries. For problems with more structure and less sparsity, higher performance is obtained by using these sparse factorization solvers.

Sparse Iterative Solution Methods: A requirement of efficient parallel implementations of the sparse iterative solvers introduced above is a decomposition of the matrix onto the processors that: 1) minimizes communication of the overlapping vector pieces in the parallel matrix–vector multiply of the iterative algorithm; 2) reduces storage of the resultant dense vector pieces on each processor; and 3) allows for load balance in storage and computation. Various parallel packages have been written that accomplish these goals to some degree [47], [48]. The mesh decompositions outlined previously can be used and integrated with the parallel iterative algorithm to solve the system.

Alternatively, a relatively simple approach that divides the sparse matrix entries among the distributed memory processors can be employed [49]. The matrix is decomposed in this implementation into row slabs of the sparse reordered system. The reordering is chosen to minimize and equalize the bandwidth of each row over the system [17], [18] [as shown in Fig. 5(b)] since the amount of data communicated in the matrix–vector multiply will depend upon the combination of equalizing the row bandwidth as well, as minimizing it. A row slab matrix decomposition strikes a balance between near perfect data and computational load balance among the processors, minimal but not perfectly optimal communication of data in the matrix–vector multiply operation, and scalability of simulating larger sized problems on greater numbers of processors. Since the right-hand side vectors in the parallel sparse matrix equation ($\mathbf{KX} = \mathbf{C}$) are the columns of $\mathbf{C}$, these columns are distributed as required by the row distribution of $\mathbf{K}$. When setting up the row-slab decomposition, $\mathbf{K}$ is split by attempting to equalize the number of nonzeros in each processor's portion of $\mathbf{K}$ (composed of consecutive rows of $\mathbf{K}$). The rows in a given processor's portion of $\mathbf{K}$ determines the rows of $\mathbf{C}$ that processor will contain. As an example, if the total number of nonzeros in $\mathbf{K}$ is nz, a loop over the rows of $\mathbf{K}$ will be executed, counting the number of nonzeros of $\mathbf{K}$ in the rows examined. When this number becomes approximately nz/P (where P is the number of processors that will be used by the matrix equation solver), the set of rows of $\mathbf{K}$ for a given processor has been determined, as has the set of rows of $\mathbf{C}$.

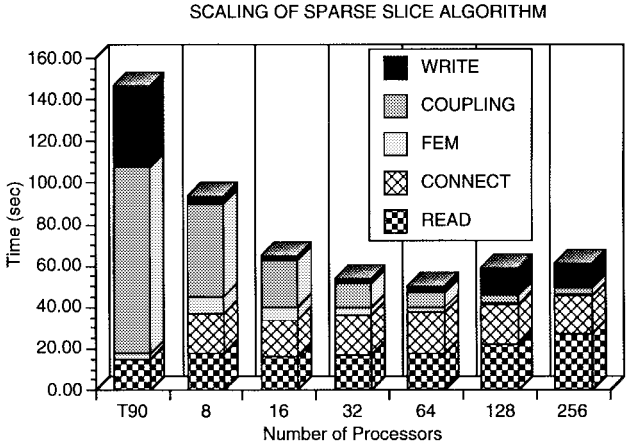


Fig. 7. Computation time and scaling for a relatively small simulation (dielectric cylinder with 43 791 edges, radius = 1 cm, height = 10 cm, permittivity = 4.0 at 5.0 GHz). The first column shows time for single processor T90. Times on T90 for CONNECT and FEM have been combined.

The matrix decomposition code used in this example consists of a number of subroutines; initially, the potentially large mesh files are read (READ), then the connectivity structure of the sparse matrix is generated and reordered (CONNECT), followed by the generation of the complex-valued entries of $\mathbf{K}$ (FEM), building the connectivity structure and filling the $\mathbf{C}$ matrix (COUPLING). Finally, the individual files containing the row slabs of $\mathbf{K}$ and the row slabs of $\mathbf{C}$ must be written to disk (WRITE). For each processor that will be used in the matrix equation solver, one file containing the appropriate parts of both the $\mathbf{K}$ and $\mathbf{C}$ matrices is written. Fig. 7 shows the performance of these routines over varying numbers of processors for a problem simulating scattering from a dielectric cylinder modeled by 43 791 edges. The parallel times on a Cray T3D are compared against the code running sequentially on one processor of a Cray T90. As mentioned above, the reordering algorithm and the algorithm generating the matrix connectivity are fundamentally sequential. These routines do not show high efficiency when using multiple processors—the time for this algorithm is basically flat—whereas for routines that can be parallelized (FEM, COUPLING, and WRITE), doubling the number of processors reduces the amount of time by a factor of approximately two. The time for reading the mesh is bound by I/O rates of the computer, and the time for writing the decomposed matrix data varies slightly for the 128 and 256 processor cases due to other users also doing I/O on the system. As will be shown in the next result, a key point of this approach to matrix decomposition is that the total time needed (less than 100 s on eight processors) is substantially less than the time needed for solving the linear system, and any inefficiencies here are less important than those in the iterative solver.

In this example, quasi-minimum residual algorithm [52] is used to solve the sparse system of equations $\mathbf{KX} = \mathbf{C}$. With the row-slab decomposition used, the machine is logically considered to be a linear array of processors, with each slab of data residing in one of the processors. Central components of the quasiminimum residual algorithm that are affected by the use of a distributed memory machine are the parallel

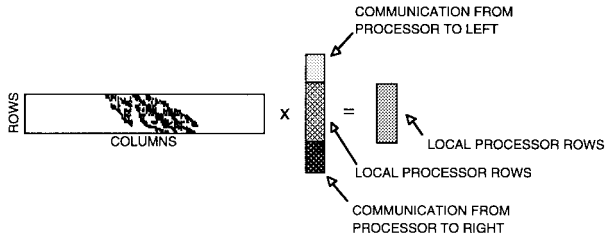


Fig. 8. Local sparse matrix-dense vector multiply graphically displayed.

sparse matrix-dense vector multiply, and dot products and norm calculations that need vector data distributed over the machine. The dominant component is the matrix-vector multiply, accounting for approximately 80% of the time required in a solution. The parallel sparse matrix-dense vector multiply involves multiplying the $\mathbf{K}$ matrix that is distributed across the processors in row slabs, each containing a roughly equal number of nonzero elements, and a dense vector $\mathbf{x}$, that is also distributed over the processors, to form a product vector $\mathbf{y}$, distributed as is $\mathbf{x}$ (Fig. 8). Since the $\mathbf{K}$ matrix has been reordered for minimum bandwidth, the minimum and maximum column indexes of the slab are known. If the piece of the dense vector $\mathbf{x}$ local to this processor has indexes within this extent of column indexes, the multiply may be done locally and the resultant vector $\mathbf{y}$ will be purely local. In general, the local row indexes of the dense vector $\mathbf{x}$ do not contain the range of column indexes; therefore, a communication step is required to obtain the portions of the multiply vector $\mathbf{x}$ required by the column indexes of the $\mathbf{K}$ matrix. This communication step only requires data from a few processors to the left and right. The exact number of processors communicating data is dependent on the row bandwidth of the local piece of $\mathbf{K}$, and the number of processors being used. In the simulations considered, the number of processors communicating data is typically one or two in each direction on scaled problems.

Shown in Fig. 9 are plots of time to convergence on different numbers of processors for five different problems (fixed size problems). The number of unknowns in the finite-element mesh and the number of columns of $\mathbf{C}$ are indicated on the plots. The quasiminimum residual algorithm was stopped when the normalized residual was reduced three orders of magnitude for each column of $\mathbf{C}$. With an initial guess being the zero vector, this results in a normalized residual of 0.1%, a value that is sufficient for this scattering problem. Given a fixed communication percentage and a fixed rate for local work, doubling the number of processors for a given problem would halve the total solution time. The curves in Fig. 9 do not drop linearly at this rate for increasing numbers of processors, because there is a decrease in the amount of work per processor while the amount of data communicated increases, causing the curves to level off.

Another factor in the performance of the parallel matrix–vector multiply is the percentage of communication. This is related to the number of processors to the left and right that each processor must communicate. It is clear that running a fixed size problem on an increasing number of processors will generate a growing amount of communication. The amount

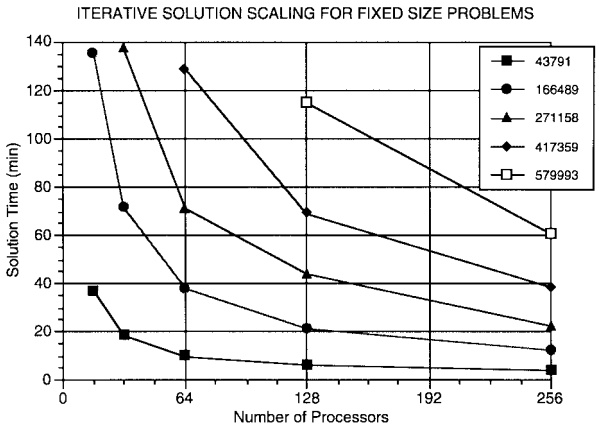


Fig. 9. Time of convergence for five different problems. The time shown is the total execution time for the solver on different numbers of processors. The $\mathbf{C}$ matrix had 116 columns in each case.

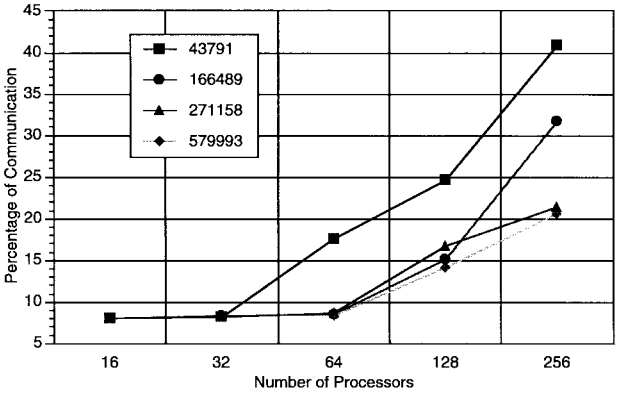


Fig. 10. Percentage of communication versus number of processors for parallel matrix–vector multiply, for four different size (number of edges) meshes of dielectric cylinder.

of communication is a function of how finely the $\mathbf{K}$ matrix is decomposed, since its maximum row bandwidth after reordering is not a function of the number of processors used in the decomposition. If the maximum row bandwidth is m and each processor in a given decomposition has approximately m rows of $\mathbf{K}$, then most processors will require one processor in each direction for communication. If the number of processors used for the distribution of $\mathbf{K}$ is doubled, each processor will have approximately $m/2$ rows of $\mathbf{K}$. Since the row bandwidth doesn't change, each processor will now require communication in each direction from two processors. But since the number of floating point operations required hasn't changed, the communication percentage should roughly double. This can be seen in Fig. 10, which shows communication percentage versus number of processors, for four problem sizes.

The row-slab decomposition is a simple means for breaking the sparse matrix equation among the processors and while the mesh-decomposition algorithms outlined above can also be used, differences between the approaches in time to solution on a parallel computer were found to be small for either approach. Two alternative mesh-decomposition schemes have been compared to the matrix-partitioning algorithm, contrasting data load balance, communication load balance, the total amount of communication, and the performance of

the local processor matrix–vector performance resulting from the specific decomposition used. The first is an algorithm termed JOSTLE [55] that uses various optimization methods to equalize the mesh partitions among the processors. The second is a multilevel graph partitioning scheme termed METIS [42]. Among the three approaches, no discernible difference was found in data and communication load balance, and in the performance of the local processor matrix–vector performance. A difference was found in the total amount of communication needed in the solution of the sparse system of equations. When normalizing the total amount of communication in the matrix partition algorithm to 1.0, the JOSTLE algorithm reduced the amount of communication to 0.26, and the METIS algorithm reduced it to 0.22. From Fig. 10, it is noted that the percentage of communication time in the complete solver is 8% for scaled-sized problems (those that fit into the minimal number of processors needed to solve the problem). It is this fraction of the total CPU time that can be reduced by the 0.26 and 0.22 fractions found using the mesh-decomposition algorithms, i.e., the total time to solve the system would be reduced by just over 6% using the METIS algorithm for mesh decomposition. It was found that the METIS and JOSTLE algorithms did produce less communication overhead as the fixed-size problem was solved on larger numbers of processors, thereby further reducing total execution time. This savings over the matrix-partitioning method is offset though, since the overall execution time decreases dramatically as seen in Fig. 9 for a fixed-size problem.

Krylov subspace methods different from the quasi-minimum residual algorithm can be coupled with a mesh- or matrix-decomposition method and used for sparse matrix solution. In [47] the conjugate gradient squared and generalized minimum residual method are used with geometric partitioning algorithms and then compared. The Krylov iterative method implementations are necessarily similar since the dominant component of the solver is the matrix–vector multiply. Parallel speedup for fixed sized problems are reported in [47] for the conjugate gradient squared and the generalized minimum residual method. The speedups are very similar to those shown in Fig. 9.

A possible means to substantially shorten the solution time in an iterative solution is the use of an effective preconditioner. The use of incomplete Cholesky preconditioners used in sequential calculations is difficult to implement in a distributed memory parallel environment due to the need for performing a forward and backward solution with each matrix multiply step in (16). On a parallel machine, these are essentially sequential operations that can give greatly reduced performance [53]. A promising alternative is to calculate an approximation to the inverse of the system, rather than a factorization of the system as is done in the incomplete Cholesky approximation. A sparse approximate inverse [54] produces a matrix with a controllable number of nonzeros that approximates the inverse of $\mathbf{K}$, and rather than calculating forward and backward solutions, it multiplies $\mathbf{K}$ at each step of the iterative algorithm. The matrix–matrix multiply can be achieved with much higher performance than the forward and backward solutions used in the incomplete factorizations.

V. Discussion

This paper presented an overview of solutions to surface-integral equation and volumetric finite-element methods on sequential and distributed memory computer architectures. Both the sequential algorithmic scalability as well as scalability on parallel computer systems were presented for current computer technology, with extrapolation to next generation technologies. A broad set of references are given. When a uniform resource locator (URL) is also referenced, it points to software which was freely available at the time this paper was written.

Acknowledgment

The authors would like to thank C. Zuffada and V. Jamnejad of the Jet Propulsion Laboratory, Pasadena, CA, for their contributions to the sequential version of the finite-element software discussed in Section IV-A.

References

[1] E. Miller, "A selective survey of computational electromagnetics," *IEEE Trans. Antennas Propagat.*, vol. 36, pp. 1281–1305, Sept. 1988.
[2] M. N. O. Sadiku, *Numerical Techniques in Electromagnetics.* Boca Raton: CRC, 1992.
[3] J. J. Dongarra, L. Grandinetti, G. R. Joubert, and J. Kowalik, Eds., "High performance computing: Technology, methods, and applications," in *Advances in Parallel Computing 10.* Amsterdam, The Netherlands: Elsevier, 1995.
[4] E. Miller, "Solving bigger problems—By decreasing the operation count and increasing the computation bandwidth," *Proc. IEEE,*, vol. 79, no. 10, pp. 1493–1504 Oct. 1991.
[5] V. Kumar, A. Grama, A. Gupta, and G. Karypis, *Introduction to Parallel Computing.* Redwood City, CA: Benjamin/Cummings, 1994; also available: http://www.cs.umn.edu/ kumar/.
[6] T. Cwik and J. Patterson, Eds., "Computational electronics and supercomputer architecture," in *Progress in Electromagnetics Research.* Cambridge, U.K.: EMW, 1993, vol. 7.
[7] R. F. Harrington, *Field Computation by Moment Method.* New York: Macmillan, 1968.
[8] A. J. Poggio and E. K. Miller, "Integral equation solutions of three-dimensional scattering problems," in *Computer Techniques for Electromagnetics*, R. Mittra, Ed. New York: Hemisphere, 1973, ch. 4.
[9] G. Golub and C. Van Loan, *Matrix Computations.* Baltimore, MD: John Hopkins Univ. Press, 1989.
[10] E. Anderson, Z. Bai, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S, Hammarling, A. McKenney, S. Ostrouchov, and D. Sorenson, *LAPACK User's Guide.* Philadelphia, PA: Soc. Indust. Appl. Mathem., 1992; also available: http://netlib2.cs.utk.edu/lapack/index.html.
[11] E. Yip and B. Dembart, "Monostatic calculations for a 3D MOM code with fast mutipole method," in *Progress in Electromagn. Res. Symp. Proc.*, Seattle, WA, July 24–28, 1995, p. 53.
[12] J. Rahola, "Solution of dense systems of linear equations in the discrete-dipole approximation," *SIAM J. Sci. Comput.*, vol. 17, no. 1, pp. 78–89, 1996.
[13] M. Sancer, R. McClary, and K. Glover, "Electromagnetic computation using parametric geometry," *Electromagn.*, vol. 10, no. 1/2, pp. 85–104, 1990.
[14] F. X. Canning, "The impedance matrix localization (IML) method for moment-method calculations," *IEEE Antennas Propagat. Mag.*, vol. 32, pp. 18–30, Oct. 1990.
[15] W. Chew, C. Lu, and Y. Wang, "Efficient computation of three-dimensional scattering of vector electromagnetic waves," *J. Opt. Soc. Amer. A*, vol. 11, no. 4, pp. 1528–1537, 1994.
[16] V. Rokhlin, "Rapid solution of integral equations of scattering theory in two dimensions," *J. Comp. Phys.*, vol. 86, pp. 414–439, Feb. 1990.
[17] R. Coifman, V. Rokhlin, and S. Wandzura, "The fast multipole method for the wave equation: A pedestrian prescription," *IEEE Antennas Propagat. Mag.*, vol. 35, pp. 7–12, June 1993.
[18] M. A. Stalzer, "A parallel fast multipole method for the Helmholtz equation," *Parallel Proc. Lett.*, vol. 5, pp. 263–274, 1995.

[19] S. M. Rao, D. R. Wilton, and A. W. Glisson, "Electromagnetic scattering by surfaces of arbitrary shape," *IEEE Trans. Antennas Propagat.*, vol. AP-30, pp. 409–418, Mar. 1982.
[20] W. Johnson, D. R. Wilton, and R. M. Sharpe, "Modeling scattering from and radiation by arbitrary shaped objects with the electric field integral equation triangular surface patch code," *Electromagn.*, vol. 10, pp. 41–64, Jan.–June 1990.
[21] J. J. Dongarra, J. Bunch, C. Moler, and G. W. Stewart, *LINPACK User's Guide.* Philadelphia, PA: Soc. Indust. Appl. Mathem., 1979; also available: http://netlib2.cs.utk.edu/linpack/index.html.
[22] J. J. Dongarra, J. Du Croz, S. Hammarling, and R. J. Hanson, "An extended set of FORTRAN basic linear algebra subprograms," *ACM Trans. Mathem. Software*, vol. 14, pp. 1–17, 1988; also available: http://netlib2.cs.utk.edu/blas/index.html.
[23] J. J. Dongarra, J. Du Croz, I. S. Duff, and S. Hammarling, "A set of Level 3 basic linear algebra subprograms," *ACM Trans. Mathem. Software*, vol. 16, pp. 1–17, 1990; also available: http://netlib2.cs.utk.edu/blas/index.html.
[24] T. Cwik, R. van de Geijn, and J. Patterson, "Application of massively parallel computation to integral equation models of electromagnetic scattering," *J. Opt. Soc. Amer.*, vol. 11, pp. 1538–1545, Apr. 1994; also available: ftp://microwave.jpl.nasa.gov/pub/PARALLEL/COMPLEX.SOLVER.
[25] J. Jin, *The Finite Element in Electromagnetics.* New York: Wiley, 1993.
[26] J. Lee and R. Mittra, "A note on the application of edge-elements for modeling 3-dimensional inhomogeneously-filled cavities," *IEEE Trans. Microwave Theory Tech.*, vol. 40, pp. 1767–1773, Sept. 1992.
[27] S. Pissanetzky, *Sparse Matrix Technology.* London, U.K.: Academic, 1984.
[28] I. S. Duff, A. M. Erisman, and J. K. Reid, *Direct Methods for Sparse Matrices.* New York: Oxford Univ. Press, 1986.
[29] E. Ng and B. Peyton, "Block sparse Cholesky algorithms on advanced uniprocessor computers," *SIAM J. Sci. Comput.*, vol. 14, no. 5, pp. 1034–1056, pp. 1034–1056, 1993.
[30] R. Barrett, M. Berry, T. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. van der Vorst, *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods.* Philadelphia, PA: Soc. Indust. Appl. Mathem., 1994; also available: http://netlib2.cs.utk.edu/templates/index.html.
[31] M. Jones and P. Plassmann, "An improved incomplete Cholesky factorization," *ACM Trans. Mathem. Software*, vol. 21, no. 1, pp. 5–17, Mar. 1995; also available: http://netlib2.cs.utk.edu/toms/740.
[32] D. O'Leary, "Parallel implementation of the block conjugate algorithm," *Parallel Comput.*, vol. 5, pp. 127–139, 1987.
[33] R. Freund and M. Malhotra, "A block-QMR algorithm for nonhermitian linear systems with multiple right-hand sides," preprint 1996.
[34] R. Mittra, O. Ramahi, A. Khebir, R. Gordon, and A. Kouki, "A review of absorbing boundary conditions for two- and three-dimensional electromagnetic scattering problems," *IEEE Trans. Magn.*, vol. 25, no. 7, pp. 3034–3040, July 1989.
[35] J.-M. Jin and V. Liepa, "Application of hybrid finite element method to electromagnetic scattering from coated cylinders," *IEEE Trans. Antennas Propagat.*, vol. 36, pp. 50–54, Jan. 1988.
[36] X. Yuan, D. Lynch, and J. Strohbehn, "Coupling of finite element and moment methods for electromagnetic scattering from inhomogenous objects," *IEEE Trans. Antennas Propagat.*, vol. 38, pp. 386–394, Mar. 1990.
[37] W. Boyse and A. Seidl, "A hybrid finite element method for near bodies of revolution," *IEEE Trans. Magn.*, vol. 27, pp. 3833–3836, Sept. 1991.
[38] T. Cwik, C. Zuffada, and V. Jamnejad, "Modeling three-dimensional scatterers using a coupled finite element-integral equation representation," *IEEE Trans. Antennas Propagat.*, vol. 44, pp. 453–459, Apr. 1996.
[39] B. Nour-Omid, A. Raefsky, and G. Lyzenga, "Solving finite element equations on concurrent computers," *Amer. Soc. Mech. Eng.*, A. Noor, Ed., pp. 291–307, 1986.
[40] A. Pothen, H. Simon, and K. Liou, "Partitioning sparse matrices with eigenvectors of graphs," *SIAM J. Matrix Anal. Appl.*, vol. 11, pp. 430–452, July 1990.
[41] B. Hendrickson and R. Leland, "An improved spectral graph partitioning algorithm for mapping parallel computations," *SIAM J. Sci. Comput.*, vol. 16, pp. 452–469, Mar. 1995.
[42] G. Karypis and V. Kumar, "A fast and high quality multilevel scheme for partitioning irregular graphs," Tech. Rep. TR 95-035, 1995, Dept. Comput. Sci., Univ. Minnesota; also available: http://www.cs.umn.edu/karypis/metis/metis.html.
[43] M. Heath, E. Ng, and B. Peyton, "Parallel algorithms for sparse linear systems," *SIAM Rev.*, vol. 33, no. 3, pp. 420–460, 1991.
[44] E. Rothberg, "Alternatives for solving sparse triangular systems on distributed-memory multi-processors," *Parallel Comput.*, vol. 21, pp. 1121–1136, 1995; also available: http://www.ssd.intel.com/appsw/ses.html.
[45] A. Gupta, G. Karypis, and V. Kumar, "Highly scalable parallel algorithms for sparse matrix factorization," Tech. Rep. TR 94-63, Dept. Comput. Sci., Univ. Minnesota, 1994.
[46] W. Dearholt, S. Castillo, and G. Hennigan, "Solution of large sparse irregular systems on a massively parallel computer," in *3rd Int. Workshop Irr. 96*, Santa Barbara, CA, Aug. 1996, pp. 49–62.
[47] J. Shadid and R. Tuminaro, "Sparse iterative algorithm software for large-scale MIMD machines: An initial discussion and implementation," *Concurrency: Practice Exper.*, vol. 4, pp. 481–497, Sept. 1992; also available: http://www.cs.sandia.gov/HPCCIT/aztec.html.
[48] M. Jones and P. Plassman, "Scalable iterative solutions of sparse linear systems," *Parallel Comput.*, vol. 20, pp. 753–773, 1994; also available: http://www.mcs.anl.gov/home/freitag/SC94demo/software/blocksolve.html.
[49] T. Cwik, D. Katz, C. Zuffada, and V. Jamnejad, "The application of scalable distributed memory computers to the finite element modeling of electromagnetic scattering," *Int. J. Numer. Methods Eng.*, to be published.
[50] A. George and J. Liu, *Computer Solution of Large Sparse Positive Definite Systems.* Englewood Cliffs, NJ: Prentice Hall, 1981.
[51] J. Lewis, "Implementation of the Gibbs–Poole–Stockmeyer and Gibbs–King algorithms," *ACM Trans. Mathem. Software*, vol. 8, pp. 180–189, 1982; also available: http://netlib2.cs.utk.edu/toms/582.
[52] R. Freund, "Conjugate gradient-type methods for linear systems with complex symmetric coefficient matrices," *SIAM J. Stat. Comput.*, vol. 13, no. 1, pp. 425–448, Jan. 1992; also available: http://netlib2.cs.utk.edu/linalg/lalqmr.
[53] E. Rothberg and A. Gupta, "Parallel ICCG on a hierarchical memory multiprocessor—Addressing the triangular solve bottleneck," *Parallel Comput.*, vol. 18, pp. 719–741, 1992.
[54] M. J. Grote and T. Huckle, "Parallel preconditioning with sparse approximate inverses," *SIAM J. Scientific Comp.*, to be published; also available: http://www-sccm.stanford.edu/Students/grote.html.
[55] C. H. Walshaw, M. Cross, and M. G. Everett, "A localized algorithm for optimizing unstructured mesh partitions," *Int. J. Supercomput. Appl. High Performance Comp.*, vol. 9, no. 4, pp. 280–295, 1995.

Tom Cwik (S'79–M'86–SM'94), for photograph and biography, see p. 459 of the April 1996 issue of this TRANSACTIONS.

Daniel S. Katz (S'88–M'95) was born in Belleville, IL, on September 11, 1966. He received the B.S., M.S., and Ph.D. degrees in electrical engineering from Northwestern University, Evanston, IL, in 1988, 1990, and 1994, respectively.

From 1993 to 1996, he was employed by Cray Research at the Jet Propulsion Laboratory, Pasadena, CA, where he worked on solving problems in computational electromagnetics through the use of massively parallel supercomputers. He has recently accepted a position with the Jet Propulsion Laboratory in the High Performance Computing Systems and Applications Group. His research interests include computational modeling of electromagnetic wave propagation, radiation, and scattering in both the time and frequency domains, as well as general algorithm development and implementation on various types of computers. The goal of his research is to develop enabling technologies for problem solving.

Dr. Katz is a member of Eta Kappa Nu and Tau Beta Pi.

Jean Patterson, photograph and biography not available at the time of publication.