5

INTEGRAL EQUATION SOLUTIONS TO RADIATION AND SCATTERING PROBLEMS USINGA COARSE-GRAINED PARALLEL PROCESSOR

T. Cwik, J. Partee, and J. Patterson

5.1 Introduction

The development of numerical techniques to model general electromagnetic systems has paralleled the advancement of computational performance. Whether the electromagnetic system is a radiating one

such as a large multi-element antenna, a scattering object such as an airplane, or a waveguiding region such as a multi-component millimeter-waveguide, the current emphasis in computational electromagnetics is towards modeling general, complex, inhomogeneous geometries. In addition, there is a continuing emphasis on modeling electrically large objects which may be many wavelengths in size, but because of a non-smooth geometry, do not allow the reliable use of asymptotic techniques.

Complementary to the development of numerical techniques is the advancement of supercomputer performance machines. Since the 1940s the peak rate of floating point operations per second (FLOPS) has increased, on average, by a factor of 10 every 7 years. Performance gains are expected to continue, but at an increasing cost due to the physical limitations inherent in sequential computing. This limitation is based on the minimum machine cycle time physically realizable and the minimum time necessary to move data to and from memory from the various functional units in the computer [1]. State-of-the-art machines currently have clock times in the 4 to 9 ns range and future systems are expected to operate at 1 nsec. It has become apparent though, that performance increases due to semiconductor technology are reaching a point of diminishing returns.

These physical limitations combined with the continued development of low-cost but relatively powerful microprocessors have motivated the development of machines that achieve supercomputer performance at a reduced cost. This is accomplished by combining many microprocessors in parallel to perform calculations concurrently. Parallel processing also offers the opportunity to develop machines which will surpass existing supercomputer architecture performance and bypass the physical limitations inherent in sequential processing. Parallel processing, of course, is not limited to organizing microprocessors. Currently, the Cray Y-MP can support eight processors and the next generation Cray machines will allow a maximum of sixteen processors. This chapter will only consider the organization of microprocessors into parallel architectures.

Since parallel architectures present an organization of processors and therefore calculations that is different from traditional sequential architectures, it has become necessary to explore different classes of problems and assess their solutions on parallel machines. Parallel architectures have been studied from the view of machine taxonomy [2,3],

machine and algorithm performance [1,4,5], and user experience over a broad range of problem classes [6-9]. This chapter specifically examines solutions of electromagnetic integral equations on parallel processors. Initially, electromagnetic modeling and machine performance are discussed. Next, the parallelization and performance of electromagnetic calculations are examined in detail on a specific machine architecture. This is followed by a discussion of the experience gained.

5.2 Electromagnetic Modeling and Computer Performance

The interplay between a physical model and its numerical simulation begins when the mathematical relations describing the physical system are discretized. The electrical size of a problem that can be simulated (in a given amount of time, to a given degree of numerical accuracy) depends on the available machine memory and speed. Generally, a direct relationship exists between the electrical problem size and the amount of machine memory necessary to complete the simulation. This relationship depends on the rate at which an unknown quantity is sampled and the method used to solve the resulting system of equations. For direct matrix solutions to integral equation formulations of radiation and scattering problems, the required storage is at least N^2 complex numbers, where N is the number of samples of the unknown function. More storage is required for other components of the calculation and code. The time to solve the assembled system of equations by decomposing the matrix into lower and upper triangular matrices (LU decomposition) is directly proportional to N^3 when N is larger than a few hundred. This time is typically the dominant component of a code when the problem is large. Other solution techniques, e.g. those that rely on iterative methods or those that use out-of-core solutions, require different amounts of machine storage and time. Because the requirements of machine storage and time can be directly analyzed, the mapping of a physical problem onto a given machine can be accurately assessed. Based on the known performance of a given machine, it is then possible to accurately predict the amount of time necessary to solve a given problem [10,11]. In the following sections, the JPL Mark IIIfp parallel processor and the mapping of calculations onto it will be discussed.

a. Parallel Architecture and the Mark IIIfp Parallel Processor

The continuing development of parallel processing in the research community has spurred the introduction of commercially available machines using state-of-the-art processors and software. In general, these machines can be grouped into categories dependent on the location of memory relative to the processors and the number of processors to be used concurrently. Memory can be shared by a number of processors, or it can be distributed, with smaller amounts of memory attached directly to a single processor. Machines from Cray and Alliant are examples of shared-memory processors. Distributed-memory machines can be divided into those machines that have large numbers of relative simple processors (16,000 to 128,000) with small amounts of memory (1–32 KBytes), and those with smaller number of processors (1–1000) and larger amounts of memory (4–32 MBytes). Machines with a large number of processors – referred to as fine-grained machines – operate synchronously, i.e., a single instruction is performed simultaneously on multiple data that has been spread over all processors in use. Thinking Machines and Goodyear build examples of these single instruction, multiple data (SIMD) computers. Machines with smaller numbers of processors – referred to as coarse-grained machines – can operate synchronously or asynchronously. Asynchronous operation occurs when multiple instructions are being performed simultaneously on multiple data spread over all processors in use. Intel, NCube, and the JPL build examples of these multiple instruction, multiple data (MIMD) computers. This categorization of parallel processing, though somewhat arbitrary, can be examined more generally through the concepts of processor clusters and hierarchical memory [1]. Here, clusters of distributed memory processors can be connected together through a common bank of shared memory. Processing within clusters is performed simultaneously, sharing data globally through the common memory banks. The three categories described above can be viewed as clusters in this general architecture, though the specific Cedar architecture described in [1] utilizes a small number (eight) of relatively powerful vector processors within a cluster.

The parallel machine used in this study is the Mark IIIfp Hypercube. A schematic diagram is shown in Fig. 5.1. The Mark IIIfp consists of at most 128 processors connected in a hypercube topology. The number of processors used is 2^d, where d is the dimension of the cube. (This study uses up to 64 processors; the second 64 are currently

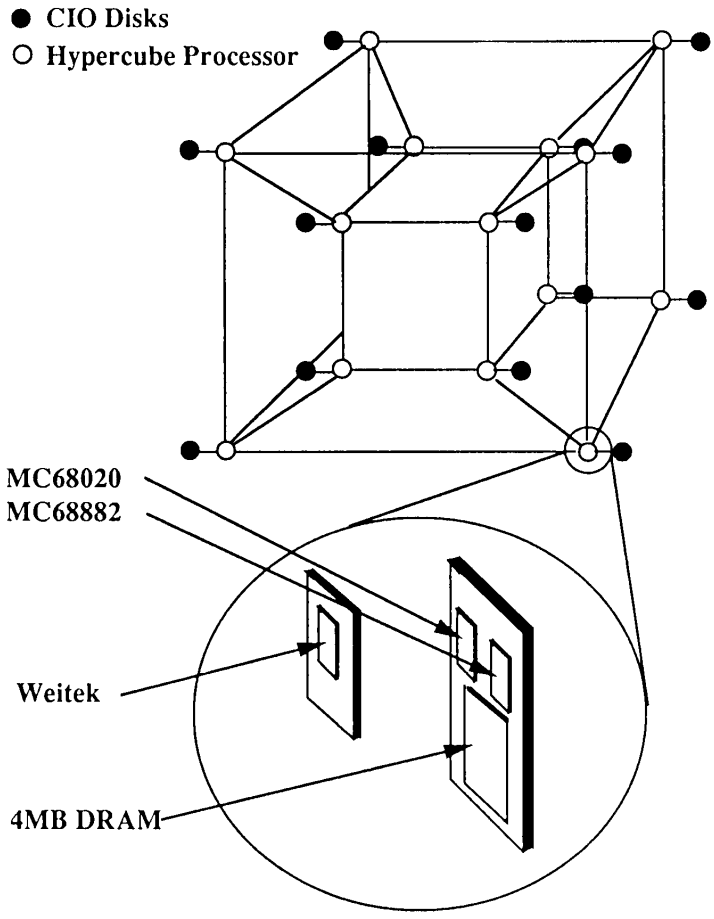


Figure 5.1 Schematic of 16-processor JPL Mark IIIfp Hypercube. This machine is attached to a host workstation.

being upgraded and are therefore unavailable.) Each processor consists of a pair of Motorola 68020 CPUs – one each for processing and communication. Floating point calculations can be performed on either a Motorola 68882 co-processor or Weitek XL series 64-bit chip set (hence the suffix *fp* for *floating point*). Dynamic RAM at each processor is 4 MBytes. Also, a number of Winchester disk drives are arranged around the processor in their own hypercube topology. These drives allow processors to write data to disk for storage, or can be used for out-of-core routines. The disk drives can operate concurrently and in parallel with

the processors, they are therefore called concurrent input and output (CIO).

One hypercube processor, numbered 0, is connected to a workstation that acts as the host processor. The user's compiled program is loaded into the hypercube from the host, and all input or output is also handled from this machine. Since the hypercube memory is distributed among processors, a message passing system is used to communicate data. This system, the crystalline operating system (CrOS), consists of a number of machine-dependent statements that are called from application programs and return information about the hypercube or allow the passing of data between processors. When using CrOS for communication, all nodes run asynchronously but align activity when communication is required. Further information on the Mark IIIfp hardware and software can be found in [12–15].

From this short description of parallel architectures, it is evident that a strategy that uses all processors efficiently to solve problems must be developed. The discretized problem should be broken up so that it is decomposed evenly into the memory attached to each processor, calculations should be equally distributed among processors, and the amount of communication between processors should be minimal. These are the topics of the next section.

b. Problem Decomposition, Load Balance and Communication

The central consideration in using a parallel processor is to decompose the discretized problem among processors so that the storage and computational load are balanced, and the amount of communication is minimal. When this is not handled properly, machine 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 incorrectly, 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. Sections 5.3 and 5.4 will outline simple methods of problem decomposition. Indeed, as shown in [8], most physical problems naturally decompose onto parallel processors. Since communication time is a factor that lowers the efficiency of an algorithm, the ratio of time needed to perform a calculation to that of communicating data between processors should be maximum. Due to hardware and software advances, the ratio of calcu-

lation to communication time is typically large for the Mark IIIfp as well as for other machines. Understanding the effect of the amount of communication, as well as the overall size of a problem relative to the hypercube memory in use, is the subject of the next section.

c. Gauging Performance

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

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

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

$$\epsilon = \frac{S}{P}$$

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 hypercube. 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 which 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 therefore remain high as successively larger problems are solved. Fixed grain problems exhibit the scalability that is a key motivator for parallel processing – successively larger problems can be mapped onto successively larger machines without a large loss of efficiency.

As an example of fixed grain and fixed problem performance, an LUD algorithm is considered. This algorithm and its performance will

be examined in more detail in Section 5.4. Scalability is plotted in Fig. 5.2 as a function of 1 to 64 processors; percentage numbers indicate efficiencies. For a fixed size problem, 339 unknowns was chosen so as to fit in one processor. It is seen that performance falls off as the size of the hypercube increases due to a decrease in the ratio of computation to communication in each processor. For fixed grain problems, the amount of data in each processor is fixed. This requires a doubling of the matrix rank (N) for each doubling of the dimension of the hypercube, i.e., $N = N_0\sqrt{P}$ where N_0 is the number of unknowns in the first processor, and P is the number of processors in use. For this example N_0 is 339. Successively larger rank matrices are decomposed at the machine efficiencies shown. As further explained in Section 5.4, a loss of efficiency is found, decreasing to 74% at 64 processors, due to required communication overhead in the algorithm. Among algorithms used in electromagnetic calculations, the 74% efficiency has been found to be generally at the low end of performance [13–15].

With the concepts of decomposition, load balance and communication developed, different solutions to scattering and radiation problems on the Mark IIIfp can now be considered.

5.3 Parallelization for Small Electromagnetic Simulations

The degree to which a code is parallelized on a given concurrent machine is dependent upon the available memory at each processor. When the amount of memory necessary to model the physical problem is less than the single processor memory, and a number of excitations, geometries, or output parameters must be varied, a particularly efficient and simple means of parallelizing the code is possible. Rather than distributing the small memory problem over all available processors and suffering the loss of efficiency noted in the previous section, each processor can execute the identical code for varying excitations, geometries or output parameters. Because there is no communication between processors except to read the input specific to the individual processor and write the individual processor's output, essentially 100% efficiency is found. Changes in existing code are minimal - as little as a few lines. Because of this simplicity and efficiency of use, this method has been termed "trivial parallelization," or "embarrassingly parallel" by workers in the field.

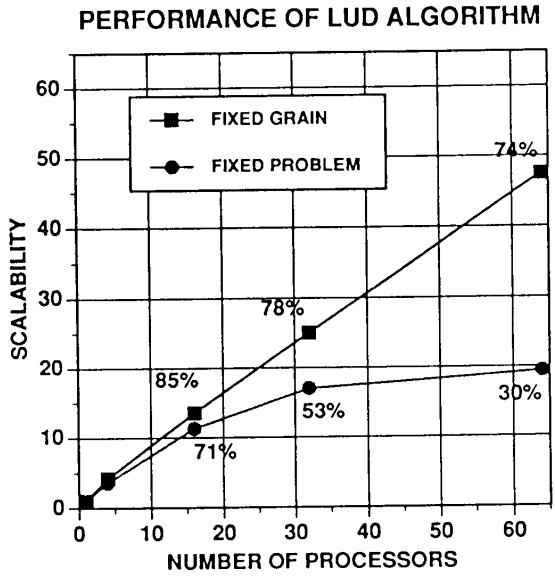


Figure 5.2 Scalability of LUD algorithm. Percentage numbers indicate efficiency.

Depending on the amount of memory at each processor and the problem size being considered, the method of trivial parallelization can be extremely useful. The Mark IIIfp has 4 Mbytes of dynamic RAM at each processor. This enables the storage of approximately an order 400 complex, double precision matrix. (Dynamic RAM at each processor must hold the executable code, necessary libraries and all storage declared in the program.) Equivalently, codes using iterative solvers which may use less storage to solve larger systems of equations, or methods other than integral equation techniques such as finite element or finite difference time domain methods, can be parallelized in this manner as long as total code memory requirements are less than the single processor memory.

The method of trivial parallelization is illustrated in Fig. 5.3. A reflection coefficient is to be calculated over a spectrum of frequencies. Each processor of the hypercube executes the identical code, accepting as input a different excitation frequency. After calculations are performed in each processor, the reflection coefficient from each processor is output. The input can be broken up any number of ways. For ex-

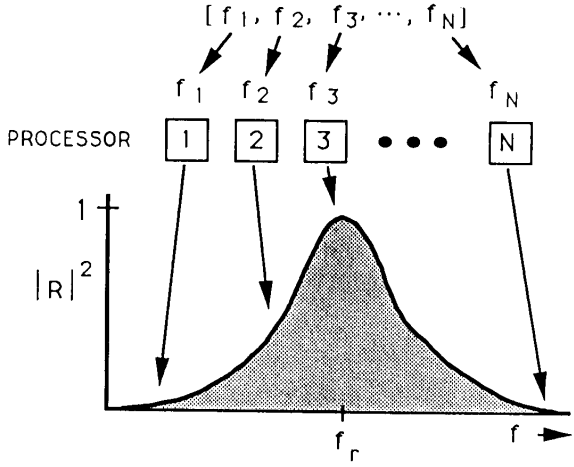


Figure 5.3 Illustration of trivial parallelization for calculation of reflection coefficient.

ample, using a 128 processor machine, a problem can be studied for four different geometries at each of 32 frequencies. Similarly, for a code which sequentially executes for varying output parameters, such as a physical optics calculation, 128 different field observation points can be calculated concurrently. In the following sections, results of two different codes which have been parallelized using this method are presented.

a. Parallelization of a Small MoM Simulation

The *PATCH* [16] method of moments code, a discretization of the electric field integral equation (EFIE) for conducting objects of arbitrary shape, is the first example of trivial parallelization considered. An object is modeled by triangular patches which conform to the surface of the object. Currents on the object are similarly modeled by pairs of the sub-domain triangular patches, a technique that results in a current representation free of line or point charges. A detailed development of the numerical techniques used in the code are found in [17], and the version of the code used in this study is described in [16]. The code models both scattering and radiating objects.

Solid modeling of the physical object is accomplished using a mesh generator which tessellates the surface into the sub-domain triangular patches. Additional input information about the excitation field, sym-

metry planes, loading, etc., and output parameter calculations such as near or far-fields (radar cross-sections), equivalent circuits, etc., are grouped into an input file to be used with *PATCH*. After execution, an output file for the given input set is generated.

Trivial parallelization proceeds by loading the executable code into each processor of the hypercube. In this example, the RCS of a square plate is to be calculated for a number of frequencies. The only FORTRAN coding necessary to pick off the input frequency specific to the individual processor the code resides in, and communicate the output RCS calculation to the host processor, is as follows:

Program PATCH

```
C DECLARATIONS SPECIFIC TO HYPERCUBE CODE
  integer env(7), node, nprocs
  open(5, file = 'input')
  open(6, file = 'output')
C GET HYPERCUBE ENVIRONMENT
  call kparam(env)
  node = env(2)
  nprocs = env(3)
C READ FREQUENCIES IN SINGLE MODE AND PICK OFF ONE
  NEEDED
  call kfsingl(5)
  do 5 ifreq = 0,nprocs - 1
  read(5,*) freqin
  if(ifreq .eq. node) freq = freqin
5 continue

C PATCH CODE

C WRITE OUTPUT IN MULTIPLE MODE
  call kfmulti(6)
  write(6,*) node, freq, rcs
  stop
  end
```

These lines are the additions necessary to complete the calculations

concurrently. Subroutine kparam is the Mark IIIfp Hypercube specific routine that returns the machine environment in array *env*. This includes the individual processor number the code resides in (*node*), and the total number of processors in use (*nprocs*). The input file, input, contains all necessary input including the set of frequencies. This file is set to 'single mode' by *call kfsingl(5)*, which causes all processors to read input simultaneously. The individual frequency for the processor in use is then picked off, and calculations of current on the plate and RCS are performed concurrently in each processor. Results are written to file output, which has been set to 'multiple mode' by *call kfmulti(6)*. In this mode, the processor number, frequency, and RCS are written to output, in sequence by processor number; i.e., file output will contain nprocs lines, each with these three different output variables. If more calculations are needed in each processor, more frequencies can be read and the calculations completed sequentially for the frequencies within each processor.

The results of these calculations are shown in Figs. 5.4a and 5.4b. Scattering by a square plate modeled by 408 unknowns is considered. A plane wave is incident normally on the plate and the backscatter RCS is calculated for plate sizes a, from 0.1λ to 1.0λ. This range is divided into 64 frequencies for use in 1 to 64 processors of the Mark IIIfp Hypercube. The RCS calculation is shown in Fig. 5.4a and performance results for both the Mark IIIfp and the Cray X-MP/18[1] in Fig. 5.4b. Measurements in Fig. 5.4a are from [17]. Optimized Cray library routines *CGECO* and *CGESL* for the LU decomposition and backward/forward substitutions, respectively, were linked to the Cray version of *PATCH*. No further optimization was attempted. The hypercube version used routines *LUDCMP* and *LUBKSB* [18] modified for complex matrices. As in the Cray version, no further optimization was attempted. Performance of the hypercube version of *PATCH* shows constant CPU time (8.1 minutes) over the entire range of calculations. Because of the minimal overhead in communicating the RCS values to the host processor, as expected, this curve is flat over all sizes of the hypercube. Because the Cray version executes sequentially, looping over all frequencies after reading the input, the CPU time necessary for

[1] 9.5 ns clock, UNICOS 5.1 operating system, CFT77 3.0 FORTRAN compiler, SN 320 (*JPL-CRAY*).

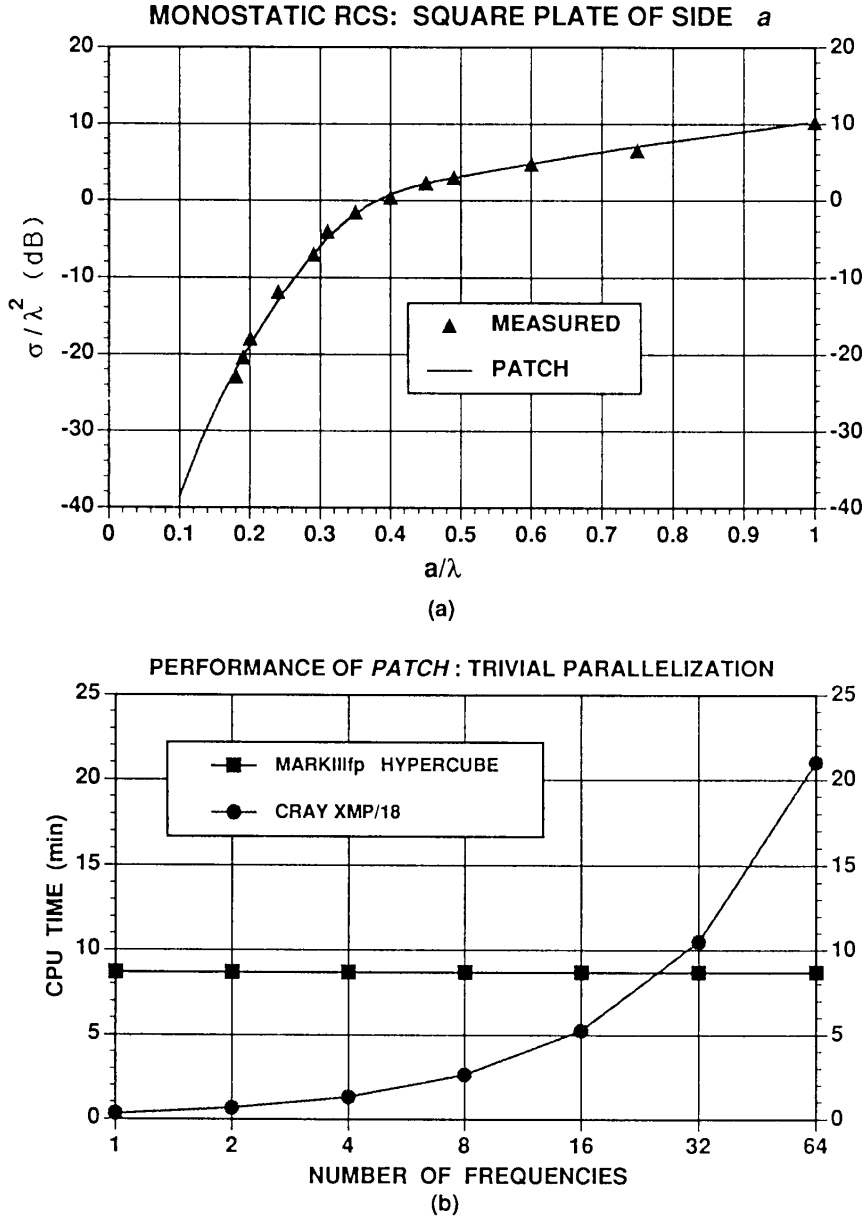


Figure 5.4 Trivial parallelization results. (a) RCS of square plate. (b) Performance of MarK IIIfp and Cray X-MP/18.

solution constantly increases. The Mark IIIfp requires less CPU time to complete these calculations, compared to the Cray, when more than 16 processors were used.

b. Parallelization of Iterative Based Simulations

In the previous MoM example, the assembled matrix was factored using a Gaussian decomposition algorithm that required a fixed number of operations dependent on the rank of the matrix. Because the matrix in all processors had the same rank, the CPU time necessary to complete the calculations was constant over all processors. In the following example of trivial parallelization, an iterative solution is used – the conjugate gradient algorithm – that requires differing amounts of computation in each processor. The amount of time necessary to complete the calculations will depend on the number of iterations needed to converge to a solution in the individual processor.

The problem considered is that of calculating the reflection and transmission coefficients of a frequency selective screen (FSS) consisting of a two-dimensional array of conducting patches on a dielectric substrate. When modeling this structure, the EFIE specific to the infinite array is constructed. This equation is solved using the conjugate gradient iterative method, yielding the discretized currents on the metal patches. Reflection and transmission coefficients of the FSS are calculated directly from this current. A detailed description of the formulation and solution method is found in [19,20].

The specific FSS geometry considered is a rectangular square patch array of period 1.0 cm in each dimension. The square patch has dimension 0.5 cm on a side, and resides on a 0.3-cm dielectric substrate of relative dielectric constant 2.2. Calculations were completed for four angles of incidence in θ $(0°, 15°, 30°, 45°)$, at each of 16 frequencies (14 – 29 GHz in 1-GHz increments), for a total of 64 calculations. Half of the reflection coefficients calculated are shown in Fig. 5.5. The performance of these calculations on the Mark IIIfp and Cray X-MP/18 is shown in Fig.5.6a for a number of excitations increasing from 1 to 64. Because this code is heavily dependent on a fast Fourier transform algorithm, the Cray library routine *CFFT* was linked to the code and used. The CPU time needed by the Mark IIIfp to complete the calculations is not constant, as in the previous example, but increases with the number of excitations and then remains constant after 16 excitations. This time dependence is clearly seen in Fig. 5.6b, which is a plot of the

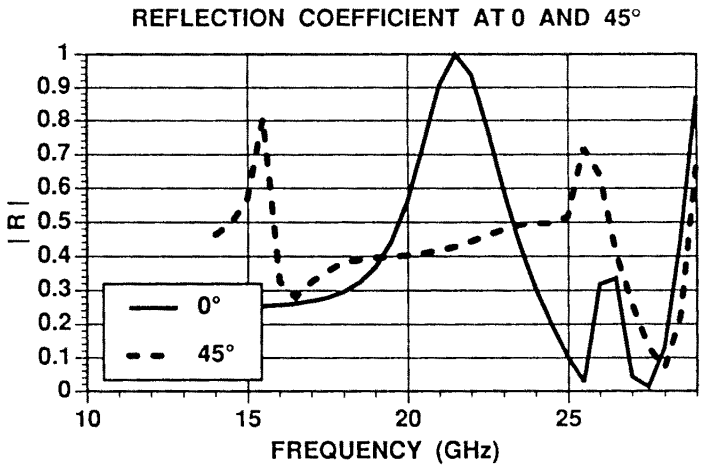
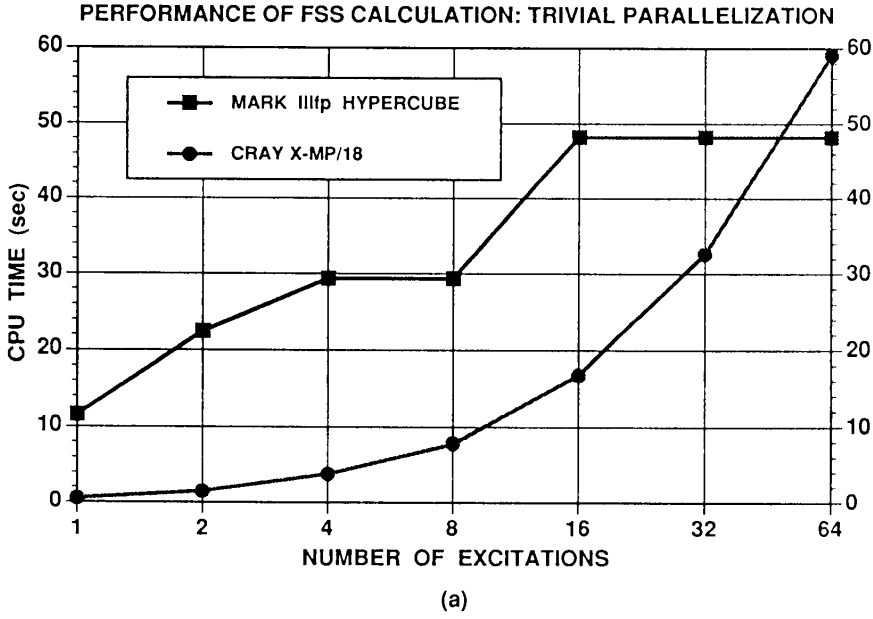
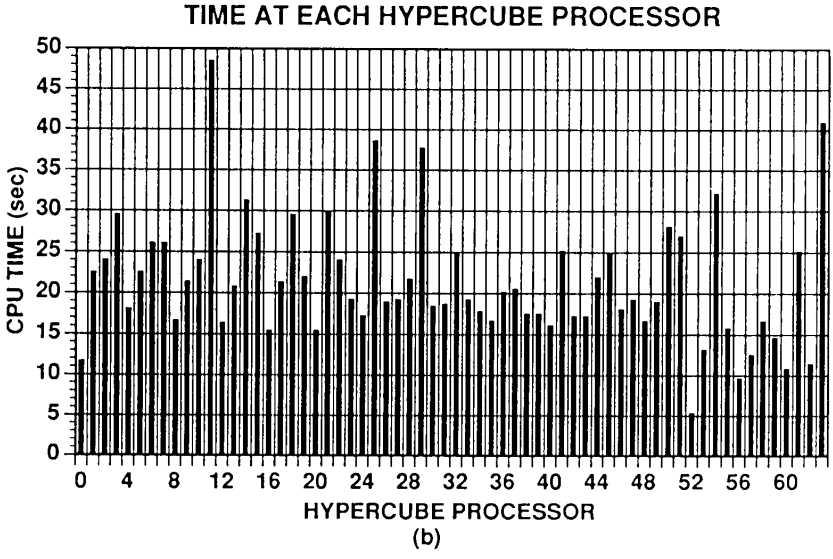


Figure 5.5 Results of FSS calculations over 64 processors. Only half of the calculations are shown.

CPU time for each of the 64 processors, and illustrates a property specific to trivial parallelization. Since a processor of the hypercube is unavailable for further computations once its calculations are completed and other processors are in use, the times in Fig. 5.6a are the maximum times needed for the calculations in each of the seven hypercube sizes (1,2,4,8,16,32,64 processors). The maximum CPU time (node 12) corresponds to the excitation at 16 GHz, $\theta = 45°$, which is at a resonance of the screen, and which causes slow convergence of the conjugate gradient algorithm. This time is therefore the maximum CPU time when more than 12 nodes are in use. Any other resonant points will also require about this amount of CPU time for convergence since the algorithm is set to terminate after a fixed maximum number of iterations. Therefore, the maximum CPU time needed by the machine will be close to the 48 seconds shown, whereas the CPU time needed to complete the calculations using the Cray will continue to increase. Fig. 5.6b also shows a general loss of efficiency due to some processors standing idle while others are completing calculations. This necessitates 64 processors to complete the calculations in less time than the Cray X-MP/18 for this code.



(a)



(b)

Figure 5.6 Results of trivial parallelization of FSS calculations. (a) Performance results of Mark IIIfp and Cray X-MP/18. (b) Mark IIIfp individual processor times.

5.4 Parallelization for Large Electromagnetic Simulations

The previous section contained examples of a simple and powerful means to parallelize electromagnetic calculations on coarse-grained machines. Problems of load balance and communication were implicitly solved by loading an identical copy of the sequential code into each processor and performing independent calculations at each node. The first example, a MoM calculation, exhibited complete load balance of computation and data over all processors. The second example, calculations using an iterative solver, showed that even though data was distributed evenly over all processors, the amount of computation in each processor varied. Both examples exhibited virtually zero communication overhead. When electrically large problems are to be solved, the method of "trivial parallelization" cannot be used and the issues of load balance and communication overhead must be addressed directly. This is the subject of the following sections.

a. Parallelization of MoM Code Components: Load Balance and Communication

The *PATCH* code, trivially parallelized in Section 5.3a, is now used for modeling electrically large objects. The central issue involved in the parallelization of *PATCH* (as for any MoM code) so that it runs efficiently on a coarse-grained parallel processor, is the decomposition of the impedance matrix. This matrix, which is now too large to fit into a single processor's memory, must be decomposed into parts that reside in each node. The decomposition should maintain a balance of the storage of matrix elements among processors, and equalize the amount of computation performed by each processor when the matrix equation is solved. Indeed, the decomposition of the matrix is directly linked to the concurrent solver used. For a direct LU factorization with partial pivoting used for numerical stability, the matrix generally can be decomposed in three ways. First, decomposition by rows, where a number of rows of the matrix will reside in a processor; second, decomposition by columns, where a number of columns of the matrix reside in a processor; and third, block decomposition, where a block of the matrix resides in a processor. It is possible to implement a pivoting strategy in each decomposition – pivoting by rows or columns in each of the first two decompositions, and pivoting within blocks in the third. These algorithms, as well as others and performance tradeoffs,

can be found in [21-23]. In the parallel implementation of *PATCH* on the Mark IIIfp, decomposition by rows with row pivoting is used. Similarly, decomposition by columns with column pivoting was performed on an earlier version of *PATCH* and is outlined in [24]. This study was completed using the Intel iPSC hypercube on up to 32 nodes and reached conclusions which are similar to those in this study.

Once the method of matrix decomposition and the solver have been chosen, parallelization of the code can proceed. *PATCH* is divided into the standard MoM code components: discretization of the geometry into induced current basis functions, matrix fill, matrix factorization and solution, and finally, field computation from the induced currents. When parallelizing an existing sequential code, as is being done with *PATCH*, it is useful to examine each block to assess code structure and relative amount of CPU time used. This provides a benchmark to assess performance and initiate a strategy for parallelization. Fig. 5.7 is a chart of the time required by the Cray Y-MP/8128 (no multi-tasking used) to complete calculations for each component of *PATCH*. Four successively larger problems—nearly doubling the matrix size (N) at size each step—were considered. As in the code used in 5.3a, Cray libraries *CGECO* and *CGESL* were used for matrix factorization and solution. CPU time dependence on the number of unknowns was found from examining the algorithms used in *PATCH*. The performance was found from the Cray performance monitor [25] and is shown in Table 5.1. Although the matrix fill component is proportional to N, [2] and the factorization proportional to N^3, the fill dominates because of the poor performance achieved in this portion of the code. It is possible that this version of *PATCH* could be optimized for vector operations. The geometry portion is ultimately proportional to N^3, although this dependence has not yet been found at 2650 unknowns. Therefore, the N^2 dependence with a large constant is also shown. The times for the matrix solve component are very small and are thus buried between the matrix factor and field calculation components of the code on this chart. Parallelization of these components now proceeds.

[2] 60-ns clock, UNICOS 5.1 operating system, CFT77 4.0 FORTRAN compiler, SN 1030, (*REYNOLDS*).

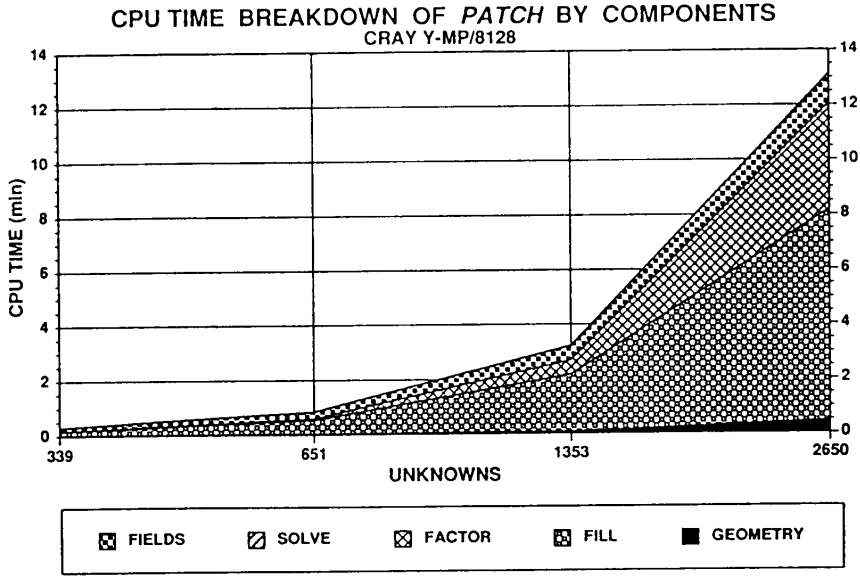


Figure 5.7 Breakdown of PATCH by components on Cray Y-MP/8128. Matrix solve time is buried between the matrix factorization and field calculation components

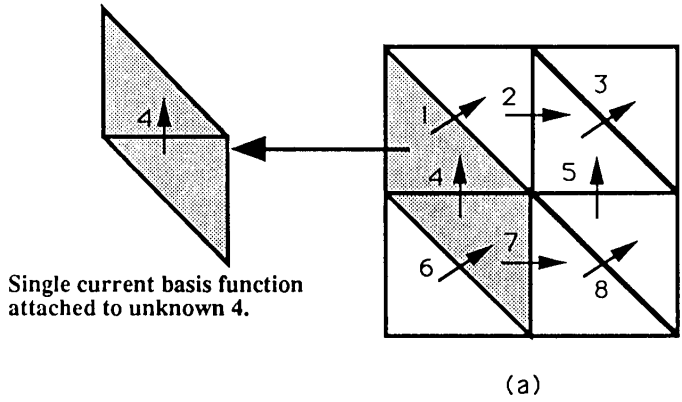
Code Component	Unkown Dependence	Performance (MFLOPS)
Geometry	$N^2 + cN^3$	—
Fill	N^2	7
Factor	N^3	212
Solve	N^2	220
Fields	N	11

Table 5.1 PATCH Code Performance

GEOMETRY. The geometry portion of *PATCH* uses a preprocessor mesh generator to model the solid body scatterer. Standard packages are used to generate input for *PATCH.* The model of the scatterer is tesselated by the preprocessor into a number of triangular patches defined by nodal points, edges and faces. This information is used by the geometry routines of the code to create a set of basis functions to model the induced currents. The geometry routines create lists of connectivity data between triangular faces and edges, the number of triangles attached to an edge, and surface normals. Disjoint bodies are also handled in this section, as well as writing the geometry data to output files.

From Fig. 5.7, it was seen that the geometry section requires a small fraction of the total code time, even at the low MFlops performance measured. Furthermore, the amount of data generated is small compared to the impedance matrix that will be generated in the fill component of the code. Because of these two reasons, the geometry section was not parallelized at this time. Each processor will complete identical calculations and store copies of all geometry data in each node. Therefore, when the matrix elements are computed, each processor will have access to all necessary data.

MATRIX FILL. In the matrix fill portion of *PATCH*, the discretized EFIE is solved by using the method of moments to form a linear system of equations whose solution gives the complex amplitude coefficients of the basis functions. As shown in Fig. 5.8a, the unknown coefficients are the amplitudes of the current directed normal to the edge of a triangle, and the spatial form of the current is represented by an interpolation function over the two triangular faces attached to that edge. The interpolation function provides a current representation free of line or point charges across patch boundaries. A triangular patch can be associated with one or more current basis functions, depending on whether or not one of its edges corresponds to an edge of the scatterer where the normal component of the current is zero, or if the patch is associated with corners or multiple intersecting parts of the scatterer. The testing procedure used to create elements of the impedance matrix integrates the field due to a basis function along a piecewise-constant path from the center of a match triangle to its edge, and onto the center of the adjacent triangle (Fig. 5.8b). The line integral is approximated by calculations performed at the center of each triangular patch to simplify the procedure. It is noted that since the testing function



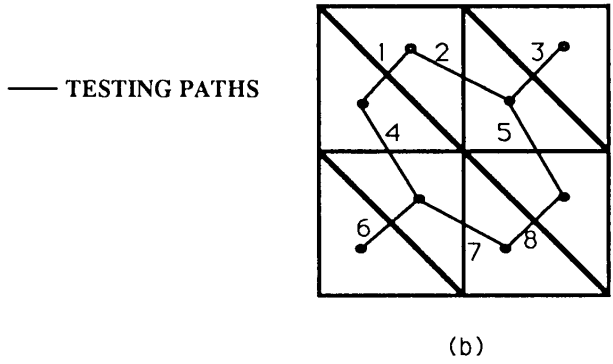


Figure 5.8 (a) Triangle basis functions. (b) Testing paths.

is not identical to the current basis function, a Galerkin method does not result, and no symmetry in the impedance matrix is found. It is also noted that since the testing procedure approximates a Galerkin method, the resulting matrix is approximately symmetric.

The current basis functions associated with a triangular patch correspond to the columns of the impedance matrix and the testing patches correspond to the rows. Ideally, to simply parallelize the matrix fill portion of the code, the fill algorithm would loop over row and column indices, performing the integrations necessary to compute matrix elements. Since, as noted above, the matrix is to be decomposed

by rows, and the calculations of any one element are independent of other elements, the algorithm need only proceed if the row being considered belongs in the processor doing the computations. Otherwise, the algorithm simply jumps to the next row. In the fill algorithm of *PATCH*, this would directly correspond to looping over all pairs of edges associated with current and match functions. In order to speed the sequential execution of the fill algorithm, it was initially recognized that calculations performed at the center of a patch are common to the three edges. It was efficient to compute the integrals associated with a triangular patch and distribute the results over elements of the matrix associated with the three edges of the patch. The fill algorithm of *PATCH* therefore loops over pairs of current and match triangular patches.

To complete the parallel decomposition, rows of the matrix were dealt out to processors in "card dealing fashion". The first row resides in the first processor, the second in the second, and so on until the nprocs row resides in the last processor of the hypercube (nprocs is the number of processors in use). The next row is stored in the first processor and the "card dealing" continues. This method allows for nearly total load balance—any processor will have at most one extra row compared to all other processors. Since the outer loop of the *PATCH* fill algorithm is over current patches associated with columns of the matrix, and the inner loop is over match triangles associated with rows of the matrix, performing a check within each processor that finds the row number for the element being calculated was inefficient. This is because the row number of an element being calculated is found in the inner loop of the algorithm and duplicate calculations are made for various current patches associated with the matrix columns, or outer loop. An interchange of loops, moving the match triangle loop calculations to the outer loop was therefore completed—a relatively simple process. The check is made in the outer loop, and if any one of its three edges will contribute to the matrix elements of the row residing in the processor performing the computation, calculations continue. Otherwise the algorithm jumps to the next match triangle. An amount of redundancy develops at this stage since calculations not contributing to the edges associated with the processor performing calculations are thrown out. They are being duplicated in the processors where they are used. This decreases the parallel performance of the fill portion of the code, as will be shown in the performance section.

MATRIX FACTORIZATION. The factorization algorithm used in *PATCH* is a row-based variant of Gaussian elimination with partial pivoting. This technique produces upper and lower triangular matrices that must be solved using both forward and backward substitution.

The elimination subroutine receives the matrix decomposed by rows. The first task in each elimination step is to determine in which row (and therefore in which processor) the maximum element of a column resides (Fig. 5.9). The maximum is first determined locally within each processor (each processor has more than one row), then the processors exchange maximum elements to determine which element is the global maximum. At each step, the k^{th} processor is responsible for the elimination row, and, if the designated processor does not have the row with the maximum element, an exchange is made. This partial pivoting is accomplished via a global broadcast call which sends a copy of the elimination row to each processor. Broadcast is a Mark IIIfp specific function that allows one node to communicate data to all other processors. When broadcast is called, the k^{th} processor will send the specified row and all other processors will receive a copy. This row is used by each processor to locally complete the remainder of the elimination step.

MATRIX SOLUTION. Currently, the solution subroutine used to perform the backward and forward substitution works over the distributed factored matrix, but it is essentially a sequential algorithm. If a processor is responsible for the substitution step associated with a specific row, the calculations proceed as normal. When the processor is finished, it uses a broadcast to communicate the updated right-hand side to all processors. If a processor is not responsible for a row, it waits for the global broadcast of the updated right-hand side before proceeding.

FIELD COMPUTATION. With the major part of computations and parallelization completed, all that remains is to calculate observational quantities associated with the scatterer. In *PATCH*, these are mainly the near and far-fields, and associated radar cross-section. Field calculations are completed by performing the forward integration of the now-known currents and appropriate Green's function evaluated at given near or far-field observation points. Because of the discretization of the current, this integration is a simple sum of the individual field components due to the current basis functions. The parallelization was completed by breaking the sum into parts equally distributed

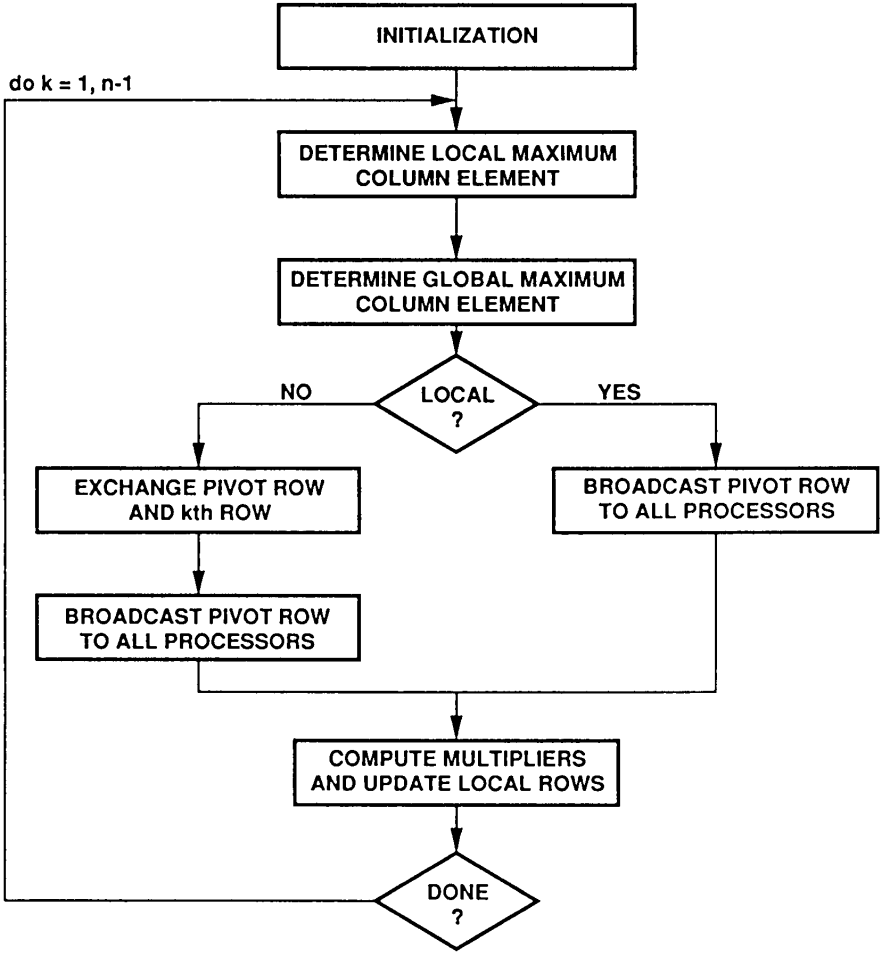


Figure 5.9 Parallel Gaussian factorization algorithm.

over all processors. Since the solution vector has been distributed to all processors, the decomposition is rudimentary–each processor calculates field components due to current basis functions associated with triangular patches over *nfaces*/*nprocs* patches (*nfaces* is the total number

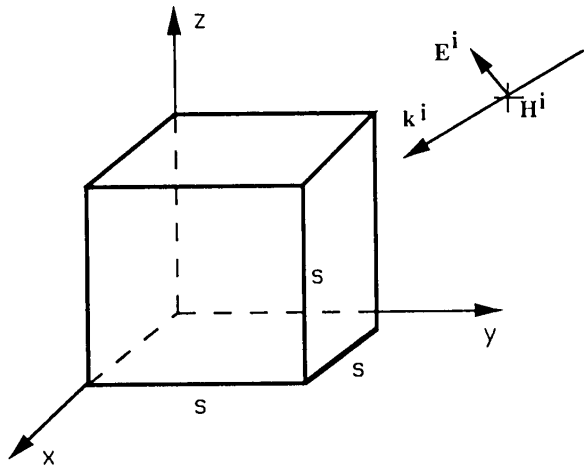


Figure 5.10 Geometry of perfectly conducting cube scatterer.

of patches). The hypercube function *kcombi* is then called to combine field components from all processors. Function *kcombi* accepts as input the field component from a processor and returns the combination of field components from all processors. How the combination is formed is defined by a second function, in this case a simple addition.

b. Parallel Performance – Scaled Problem Size

The performance of the parallelized version of *PATCH* is now presented for a perfectly conducting cube of side s (Fig. 5.10) which was scaled proportionally to the number of processors in use. The incident field directed broadside to the scatterer and both monostatic and bistatic radar cross-sections (RCS) were calculated for various size cubes. When the bistatic RCS was calculated, the scattered field was calculated in the E, H and 45° planes for ϕ from 0° to 180°.

The performance of the parallelized code is initially understood from an examination of Fig. 5.11. This chart shows the time for code components as the problem size is increased. The number of unknowns nearly doubles at each new data point – each point corresponding to increasing the dimension of the hypercube by 2. Therefore, for 339, 651, 1353, and 2650 unknowns, the number of processors used was 1, 4, 16, and 64 respectively. Since storage of the matrix increases as

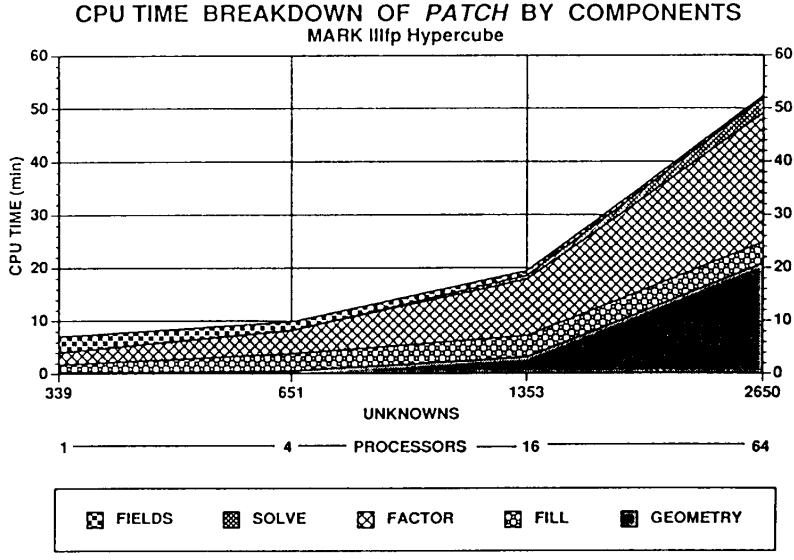


Figure 5.11 Breakdown of PATCH by components on Mark IIIfp. The number of unknowns scale with the size of the hypercube (fixed grain problem).

the square of its rank, the above increase in unknowns and increase in hypercube size corresponds to a scaling where the number of matrix elements stored in each processor is nearly constant (fixed grain size), and at the maximum storage allowed. (The matrix rank does not exactly double at each step because of how the perfectly conducting cube is discretized by the mesh generator.) It is immediately seen that the matrix factorization component is now dominant. This differs from Fig. 5.7, which showed the matrix fill component dominating when *PATCH* is executed on the Cray Y-MP. The parallelized fill component is nearly constant as the problem size increases, and the geometry component begins to increase with the problem size. Matrix solution and field calculation components are relatively small. Each component's performance is now considered separately.

GEOMETRY. The geometry portion of the code was not parallelized. For small problem sizes, this portion of the code is also small, but as the problem size increases, the time for geometry processing grows and will eventually dominate. The next stage of parallelization

would therefore be the decomposition of the geometry processing section. Due to the involved structure of this section of code, this would involve a rewrite. One possible strategy for parallelization is to distribute the discretized input geometry data set over the hypercube and allow each processor to compute the lists of data associated with its part of the input geometry data set. This information would then be communicated to all other processors so that each processor would have all geometry data for use with the matrix fill portion of the code. Full use of the hypercube could be made, and the communication portion of the algorithm would be a small fraction of the time needed for calculations.

MATRIX FILL. From Fig. 5.11, it is noted that the time to perform the calculations needed to fill the matrix is nearly constant as the problem size increases. This is expected since the number of unknowns, N, scales as the square root of the number of processors P, i.e., $N(P) = \sqrt{P}$. The time to fill the matrix is $CN(P)^2 = CN_0^2 P$ where N_0 is the number of unknowns in a single processor and C is a constant. The time to fill the matrix for a problem scaling with the size of the hypercube is therefore ideally

$$T_{FILL}(P) = \frac{CN_0^2 P}{P} = CN_0^2$$

which is the time necessary to fill the matrix when a single processor is used. The measured scalability is shown in Fig. 5.12. As outlined in Section 5.2, scalability is the ratio of the time necessary to complete calculations on a hypothetical single processor to that necessary on P processors. For the fill calculations, the scalability is

$$S_{FILL}(P) = \frac{CN^2}{T(P)} = \frac{CN_0^2 P}{T(P)} = \frac{P}{\tilde{T}(P)}$$

where the tilde denotes measured time on P processors relative to that on one processor. As noted in Fig. 5.12, performance is not 100% efficient, but is constant with the number of processors. This is due to the redundancy in computing the matrix elements as explained in the previous section. When looping over current and match triangles in each processor, redundant calculations were performed and excess information discarded. Ideally, a single integration would be performed, and, the worst case would involve three integrations. From Fig. 5.12, it is

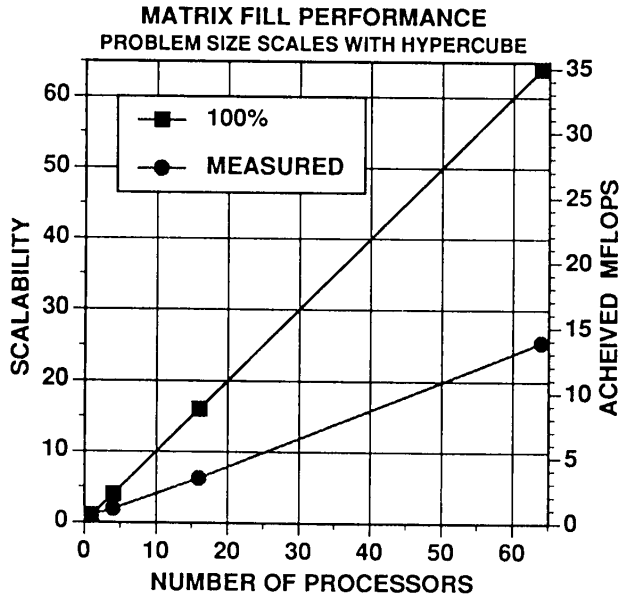


Figure 5.12 Performance of matrix fill portion of code.

calculated that the redundancy factor is 2.5, meaning that on average over the machine, 2.5 matrix element calculations are performed and discarded for each one kept. This factor will be constant on any size machine larger than several processors and is solely dependent on the nature of the fill algorithm. The elimination of this redundancy would most easily be made along with the parallelization of the geometry section of *PATCH*. Therefore, the strategy outlined in the geometry section above for parallelizing that portion of the code would be augmented to mate with a parallel fill algorithm that has no redundant calculations.

Achieved MFLOPS performance is also marked in Fig. 5.12. The number of operations in this portion of the code was found from the Cray Y-MP performance monitor. A single processor performed at approximately 0.55 MFLOPS. Achieved MFLOPS refers to actual machine performance, i.e., total time to complete the calculations, including any hypercube communication overhead.

MATRIX FACTORIZATION. From Fig. 5.11, it is noted that the time to factor the double precision, complex, non-symmetric matrix into lower and upper triangular matrices, partial pivoting included, in-

creases with problem size. Following reasoning similar to the matrix fill calculations above, the time to factor the matrix is since the algorithm scales as N^3. Spreading this calculation over P processors gives

$$T_{FACTOR}(P) = \frac{CN_0^3 P\sqrt{P}}{P} = CN_0^3\sqrt{P}$$

which increases with the square root of the number of processors. Measured scalability is shown in Fig. 5.13 and is found from the relationship

$$S_{FACTOR}(P) = \frac{CN^3}{T(P)} = \frac{CN_0^3 P\sqrt{P}}{T(P)} = \frac{P\sqrt{P}}{\tilde{T}(P)}$$

where again the tilde denotes normalizing the measured time on P processors to that on a single processor. Measured performance is now seen to depend on the number of processors, decreasing from an ideal 100% as the problem size increases. The decrease in efficiency is due to the communication overhead necessary in the process of partial pivoting. At each step of the factorization, the maximum element in each row must be communicated to all other processors and compared. If pivoting is necessary, the matrix row containing the maximum element must also be communicated to all processors. This communication therefore reduces the parallel efficiency of the algorithm. The achieved performance shows about 0.75 MFLOPS when a single processor is used.

MATRIX SOLUTION. Once the matrix is factored into lower and upper triangular matrices, the process of forward and backward substitution is performed to complete the solution. Both substitution algorithms contain loops where calculations at each stage in one processor depend on previous calculations in different processors [26]. A straightforward parallelization therefore becomes more involved. The parallel matrix solution algorithm used, essentially rewrites the solution vector to all processors as the backward and forward substitution progresses. As seen in Fig. 5.11, this portion of the code, even at 2650 unknowns, is still a small fraction of the total time.

FIELD CALCULATION. After the matrix is solved, the now-known current vector resides in each processor and secondary calculations of observable quantities can be made. RCS of the cube was calculated in the E, H, and 45° planes for θ from 0° to 180° in 1° increments. The time to complete the calculation scales as N, and therefore the time over P processors in a scaled problem is

$$T_{FIELD}(P) = \frac{CN_0\sqrt{P}}{P} = \frac{CN_0}{\sqrt{P}}$$

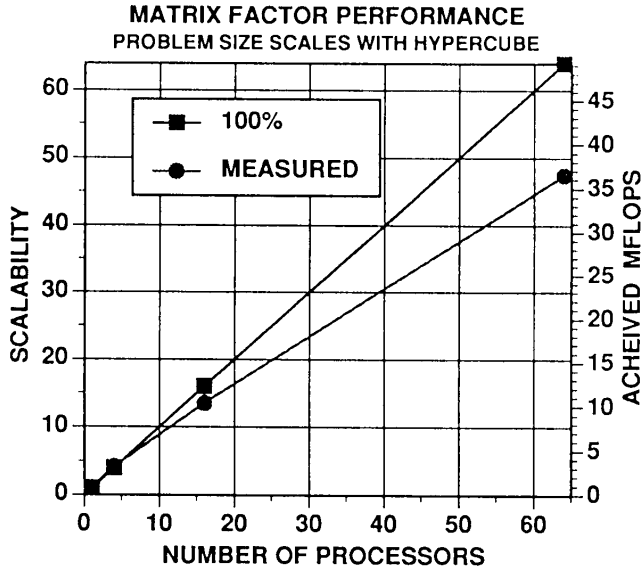


Figure 5.13 Performance of matrix factorization portion of code.

and is seen to decrease with the number of processors used.

Though the time to calculate fields decreases as the problem size increases, becoming an insignificant portion of the code, it is instructive to examine the scalability of the field calculations. Measured scalability is shown in Fig. 5.14 and is found from the relationship

$$S_{FIELD}(P) = \frac{CN}{T(P)} = \frac{CN_0\sqrt{P}}{T(P)} = \frac{\sqrt{P}}{\tilde{T}(P)}$$

where again the tilde denotes normalizing the time on P processors to that on one processor. As in factoring the matrix, a loss of efficiency is found as the problem size increases. But this loss of efficiency is not due to an increase of communication overhead, as in the factorization algorithm, since only a single communication call is needed to combine individual field calculations from all processors. The loss of efficiency here is due to the scaling of the problem and the number of operations needed to compute the portion of the field from each processor. Because the field calculation requires a loop over N current basis functions that is spread over P processors, and the number of unknowns increases as

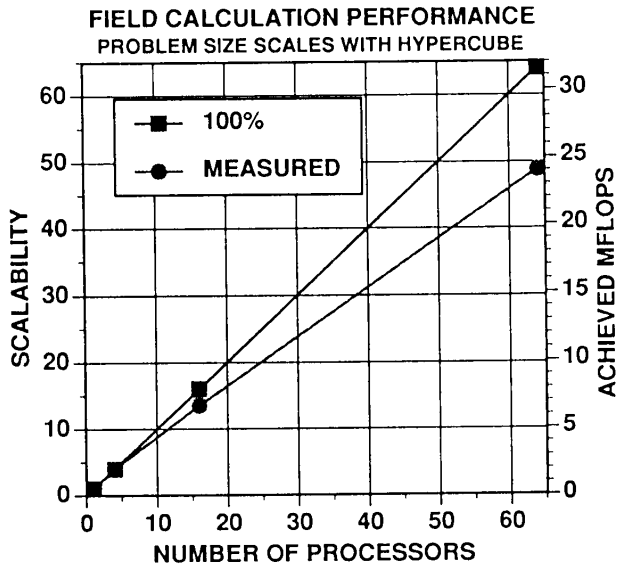


Figure 5.14 Performance of field calculation portion of code.

$\sqrt{P}$, the number of calculations in each processor decreases by $\sqrt{P}$. The sequential components of the calculation and the communication overhead are fixed; therefore a loss of efficiency results. On a single processor, 0.5 MFLOPS was achieved.

c. Parallel Performance – Fixed Problem Size

The previous section examined the performance of MoM calculations that scaled with the size of the hypercube in use. As shown in Section 5.2, this produces maximum efficiency and allows the solution of larger and larger problems as the machine size increases. However, if a fixed size problem is to be solved that does not completely fill the memory of the hypercube in use, the user would spread the problem over the maximum size machine and accept a loss of efficiency to minimize the amount of time necessary to solve the problem. This has been done for the cases of 339, 651, and 1353 unknowns and is shown in Fig. 5.15. The problem was solved on all hypercube sizes capable of storing the matrix and data. The time shown is the total CPU time necessary to complete the solution. For the smallest prob-

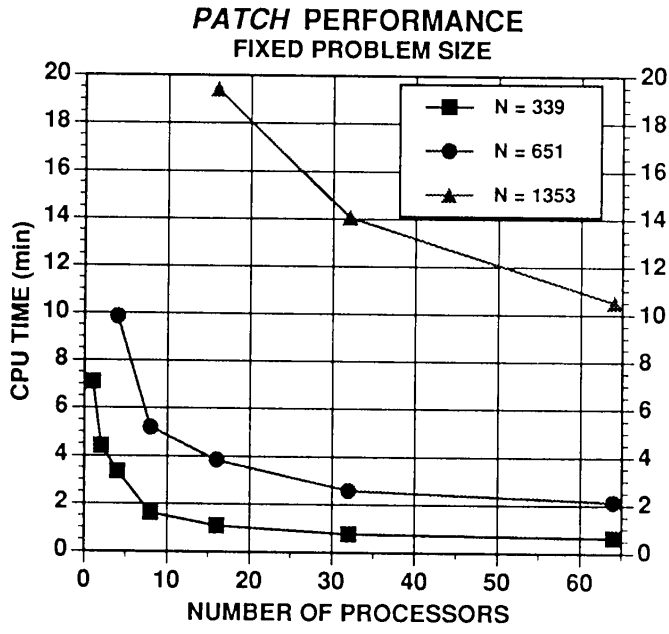


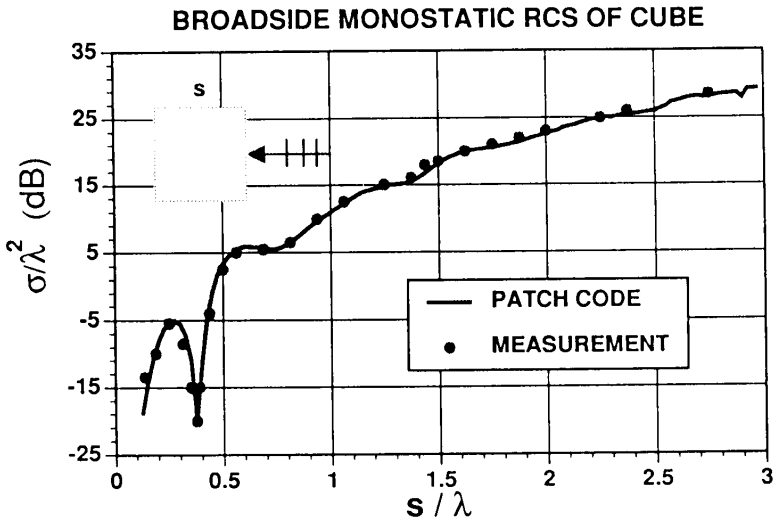
Figure 5.15 Performance of fixed size problems on the Mark IIIfp.

lem (N = 339), considerable speedup is noted through 16 processors, at which point the total time is flat for larger hypercubes. Similarly 651 unknowns flattened out at 32 processors, and 1353 unknowns has yet to reach saturation at 64 processors. The Cray Y-MP single processor time needed to solve these problems was 0.30, 0.85, and 3.23 minutes, respectively.

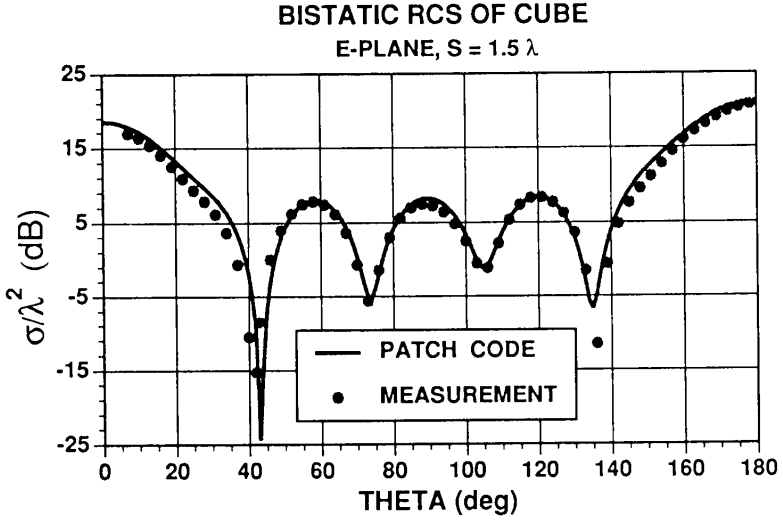
Radar cross-section calculations from the parallelized *PATCH* code are compared to measurements and shown in Fig. 5.16. Measurements of the cube were obtained from [27]. Fig. 5.16a shows normalized broadside monostatic RCS calculations as a function of s/λ. Gridding was increased until convergence of the RCS was found, and, calculations were spread over the 64 processor Mark IIIfp. Fig 5.16b shows bistatic RCS in the E-plane for a cube side dimension of 1.5λ. Again, the single calculation was spread over 64 processors.

d. Sub-Cube Parallelization

Trivial parallelization is very useful if the small problem is to be resolved many times, as in a design situation or if the system response to multiple frequencies is needed. Naturally, complete parallelization



(a)



(b)

Figure 5.16 RCS calculations and measurements. (a) Monostatic RCS. (b) Bistatic RCS.

for large problems introduces scalability, i.e., the ability to solve larger and larger problems as the hypercube size increases. If a medium sized problem is to be solved multiple times, then it is possible to apply the idea of trivial parallelization to sub-cubes of the machine. The large hypercube can be divided into sub-cubes, and the parallel code loaded into each sub-cube and executed for different input sets. Since there is virtually no communication between sub-cubes, each group of processors can be executing at maximum efficiency and the ensemble will then achieve the minimum total CPU time to complete the solution. As in the trivial parallelization cases presented previously, the sub-cubes can be divided by excitation, geometry, or output parameters.

5.5 Discussion

The central issue in using a coarse-grained machine such as the Mark IIIfp hypercube is problem decomposition. For a direct solution using a MoM technique, this means that the matrix must be decomposed among processors to achieve load balance. Since this decomposition is dependent on the matrix solver to be used, the first step in using the parallel processor was to develop a solver. With the maturity of parallel processing, solvers have become library routines where only calling arguments need be specified. The MoM matrix is therefore decomposed among processors according to the solver's instructions and computation is begun. Scalability can be high for large problems that scale with the hypercube dimension. If a fixed size problem is to be solved, performance eventually tapers off due to communication demands.

Alternatively, the method of trivial parallelization can be powerful when the problem size is small and many runs are needed. As maturing technology allows more memory to be attached to individual processors, this technique can be used for larger problems.

The motivating factors of concurrent processing – scalability of problem to processor size, and an increase in performance-to-cost ratio – continue to be justified. Interestingly, the user community, including workers engaged in computational electromagnetics, have been supplying input to manufacturers of new concurrent machines. It is therefore to be expected that concurrent processing will be an integral part of electromagnetics in the near future.

5.6 Addendum

Since the material in this chapter has been written, continued development of parallel computer systems has led to machine performance that vastly surpasses the results presented in the previous sections. Both hardware and software development has evolved greatly, allowing larger problems to be solved in smaller amounts of time. Developments in machine hardware and system software have also spurred the evolution of the computational electromagnetic algorithms presented in this chapter. In this addendum, updated material will be presented for the solution of large electromagnetic simulations using the method of moments solution. Specifically, the current LU factorization component of the integral equation solution presented in Section 5.4 will be outlined.

The matrix decomposition method outlined in Section 5.4 involved spreading the rows of the matrix among the processors in a "card dealing" fashion. This has evolved into a decomposition scheme where blocks of the matrix are computed and stored in each processor. The block decomposition is used so as to produce an extremely efficient LU factorization algorithm—one that scales nearly uniformly with machine size for scaled problems. The block decomposition is superior because it exploits matrix-matrix multiplies in the routine, and introduces a communication scheme that minimizes overhead in the message passing portion of the algorithm. The matrix-matrix multiply kernel is essential since it produces the optimum performance when using the vendor-supplied basic linear algebra subroutine (BLAS) libraries. The communication scheme is efficient because special routines communicate data between processors that have been logically mapped as a two-dimensional mesh [28].

This parallel LU factorization and solution algorithm has been used with the PATCH code previously described. The code has been run extensively on Intel iPSC Hypercube and Touchstone Delta systems. The hypercube has upwards of 64 processors, and the Delta system is an two-dimensional array (16x32) of 512 processors. Each processor has 16 Mbytes of RAM attached. Shown in Fig. 5.17 is a plot of the time to factor, and the performance of the block factorization algorithm running on the Touchstone Delta system. The problem size scales with machine size; data points are shown at 16, 64, 128, 256 and 512 processors. This corresponds to matrix orders of 3042, 5832, 8712, 12168, and 16200 unknowns, respectively. This plot can be com-

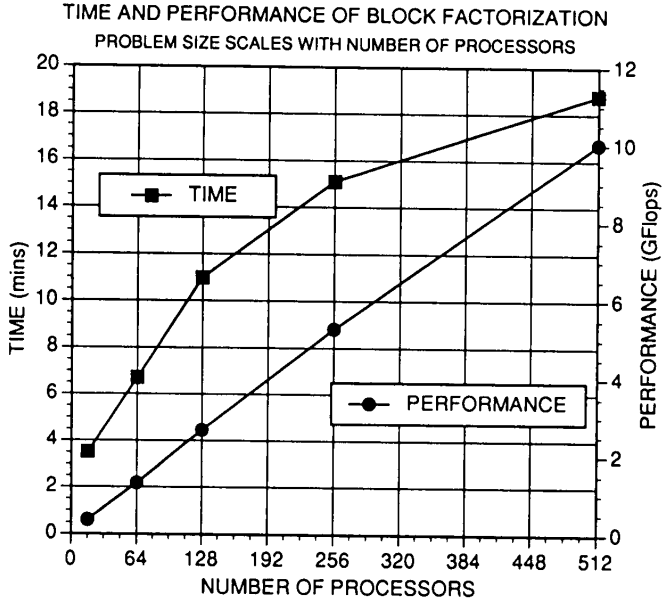


Figure 5.17 Time to factor, and performance for fixed grain problems on Intel Touchstone Delta system.

pared to Figure 5.13. Performance has increased by a factor of 278, and problem size has increased by over 6 over the 2 years separating the computations.

Acknowledgments

The authors wish to gratefully acknowledge the following people for assistance during the course of this study: William Johnson for supplying the *PATCH* code, Wilson Pearson for information regarding work on the parallelization of an earlier version of *PATCH*, and Marc Cote for the measured cube RCS. A special acknowledgment is extended to Don Wilton for discussions on the formulation and implementation of the *PATCH* code. The authors also wish to acknowledge the use of the JPL/Caltech CRAY X-MP/18 to perform parts of these calculations.

The research described in this chapter was carried out at the Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration.

This research was performed in part using the Intel Touchstone Delta System operated by Caltech on behalf of the Concurrent Supercomputing Consortium. Access to this facility was provided by NASA.

References

[1] Kuck, D. J., E. S. Davidson, D. H. Lawrie, and A. H. Sameh, "Parallel supercomputing today and the Cedar approach," *Science,* **231**, 967–974, Feb. 28, 1986.

[2] Seitz, C., "The cosmic cube," *Commun. ACM,* **28**, 22–23, 1985.

[3] Flynn, M., "Some computer organizations and their effectiveness," *IEEE Transactions on Computers,* **C-21**, No. 9, 948–960, Sept. 1972.

[4] Kumar, M., "Measuring parallelism in computation-intensive scientific/engineering applications," *IEEE Transactions on Computers,* **C-37**, No. 9, 1088–1098, Sept. 1988.

[5] Hillis, W. D., and G. L. Jr. Steele, "Data parallel algorithms," *Communications of the ACM,* **29**, No. 12, 1170–1183, Dec. 1986.

[6] Fox, G., et al., *Solving Problems on Concurrent Processor*, Prentice Hall, New Jersey, 1988.

[7] Fox, G. C., and A. Frey, "High performance parallel supercomputing application, hardware, and software issues for a teraflop computer," *Caltech Concurrent Computation Program Paper,* **C3P-451C**, Nov. 1988.

[8] Fox, G. C., "1989 - The first year of the parallel supercomputer," *Caltech Concurrent Computation Program Paper,* **C3P–769**, June 1980.

[9] Gustafson, J. L., G. R. Montry, and R. E. Benner, "Development of parallel methods for a 1024-processor hypercube," *SIAM J. Sci. Stat. Comput.* **9**, No. 4, 609–638, July 1988.

[10] Messina, P., et al., "Benchmarking advanced architecture computers," *Caltech Concurrent Computation Program Paper,* **C3P–712**, 2–53, June 1989.

[11] Dongarra, J. J., "Performance of various computers using standard linear equations software in a fortran environment," Technical Memo 23, Math and CS Div., Argonne National Lab, Jan. 29, 1989.

[12] Patterson, J. E., et al., "Parallel computation applied to electromagnetic scattering and radiation analysis," *Electromagnetics,* **10**, no. 1–2, 21–39, Jan–June 1990.

[13] Calalo, R., et al., "Hypercube matrix computation task, report for 1986-88," JPL Publication 88–31, Aug. 1988.

[14] Calalo, R., et al., "Hypercube matrix computation task, research in parallel electromagnetics, Report for 1988–89," JPL Publication, Nov. 1989.

[15] Cwik, T., et al., "Hypercube Matrix Computation Task, Research in Parallel Electromagnetics, Report for 1989–90," JPL Publication, 1990.

[16] Johnson, W., 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," *Electromagnetics,* **10**, No 1–2, 41–64, Jan–June 1990.

[17] Rao, S. M., D. R. Wilton, and A. W. Glisson, "Electromagnetic scattering by surfaces of arbitrary shape," *IEEE Trans.* **AP-30**, 409–418, May 1982.

[18] Press, W., et al., *Numerical Recipes, The Art of Scientific Computing*, Cambridge University Press, Cambridge, 1986.

[19] Cwik, T., and R. Mittra, "Scattering from a periodic array of free-standing arbitrarily shaped perfectly conducting resistive patches," *IEEE Trans.*, **AP-35**, 1226–1234, Nov. 1987.

[20] Cwik, T., and R. Mittra, "The cascade connection of planar periodic surfaces and lossy dielectric layers to form an arbitrary periodic screen," *IEEE Trans.*, **AP-35**, 1397–1405, Dec. 1987.

[21] Chu, E., and A. George, "Gaussian elimination with partial pivoting and load balancing on a multiprocessor," *Parallel Computing*, **5**, 65–74, 1987.

[22] Geist, G. A., and C. H. Romine, "LU factorization algorithms on distributed memory multiprocessor architectures," *SIAM J. Sci. Stat. Comput.*, **9**, No. 4, 639–649, July 1988.

[23] Gallivan, K. A., R. J. Plemmons, and A. H. Sameh, "Parallel algorithms for dense linear algebra computations," *SIAM Rev.*, **32**, No. 1, 54–134, March 1990.

[24] Wilton, D., and R. Sharpe, "Solution of scattering problems by the method of moments on a hypercube," Technical Report No. 87–14, McDonnell Douglas Corporation, Aug. 1987.

[25] *UNICOS Performance Utilities Reference Manual*, Cray Research, Publication No. SR-2040, May 1989.

[26] Golub, G., and C. van Loan, *Matrix Computations*, Second Edition, The John Hopkins University Press, Baltimore, 1989.

[27] Cote, M. G., M. B. Woodworth, and A. D. Yaghjian, "Scattering from the perfectly conducting cube," *IEEE Trans.*, **AP-36**, 1321–1329, Sept. 1988.

[28] van de Geijn, R., "Massively parallel LINPACK benchmark on the Intel Touchstone DELTA and iPSC/860 systems," **TR-91-28**, Dept. Comp Sci, U of Texas, 1991.