4

SOLVING PARTIAL DIFFERENTIAL EQUATIONS FOR ELECTROMAGNETIC SCATTERING PROBLEMS ON COARSE-GRAINED CONCURRENT COMPUTERS

R. D. Ferraro

4.1 Introduction

The general or "canonical" electromagnetic scattering problem is a computationally intensive task. The scatterer may consist of a collection of objects of various shapes and sizes, composed of a variety of materials. These materials may be electrically simple, such as (near) perfect electric conductors, or may be complicated anisotropic compounds which are lossy or ferromagnetic. The degree to which these properties need to be considered depends largely upon the frequency and wavelength of the incident radiation, and upon the accuracy desired for the scattered field solution. The long wavelength ($\lambda \gg L_s$, where L_s is some characteristic dimension of the scatter) and short wavelength ($\lambda \ll L_s$) limits of the scattering problem have different

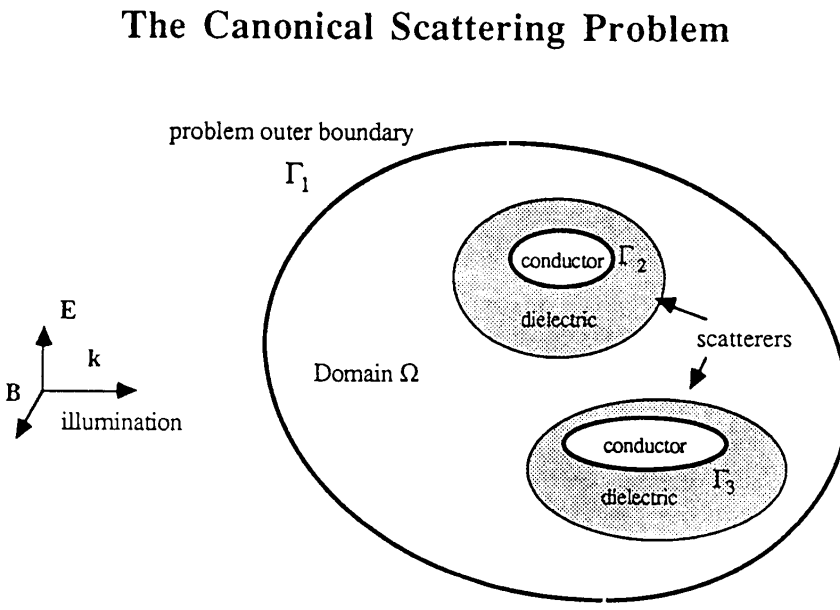

Figure 4.1 The canonical scattering problem. An incident electromagnetic wave is scattered by a collection of objects. Each object may be composed of various types of materials, including conductors and lossy anisotropic dielectrics. The objects are contained within a computational domain Ω, within which the fields are to be computed.

computational considerations than does the intermediate wavelength ($\lambda \approx L_s$) scattering problem. The method of solving each of these is generally quite different.

The difficulty in solving the intermediate wavelength problem stems from the inability to ignore complications of the scatterers' geometry and composition. A schematic representation of the canonical scattering problem is presented in Fig. 4.1. In the intermediate wavelength limit, interference from reflections of the incident field from different parts of the object may be important. Standing waves or surface waves inside dielectric coatings may modify the reflection properties, with a strong frequency dependence. Accurate solutions to the scattering problem in these cases require accurate modelling of the scatterers

themselves.

The computational solution of a partial differential equation is an attractive technique for the intermediate wavelength case in the presence of dielectrics for two reasons. Incorporation of arbitrarily complicated combinations of dielectric materials into the equations is straightforward. And the local nature of a partial differential equation approach translates into sparse systems of linear equations, even when the geometries are complicated. The approach does have the drawback of requiring a truncated computational domain with some kind of outgoing wave boundary condition, which is an additional source of error in the solution. (We will not consider non-local truncation methods such as matching to an integral equation formulation in this discussion). And the entire computational domain must be meshed at sufficient mesh density to adequately represent travelling wave solutions. The sparsity of the resulting system of linear equations saves the method, however, since the storage requirements scale linearly with the volume of the computational domain.

In this chapter, we describe the techniques used to solve Maxwell's partial differential equations on a multicomputer architecture. We have implemented two well-known methods of solving Maxwell's equations on the Caltech/JPL Mark IIIfp Hypercube [1] multicomputer. The first is a Finite Difference Time Domain method, which explicitly follows the evolution of an incident electromagnetic wave in time as it impinges upon the scatterers. The second is a Finite Element frequency domain method, which has the virtue of being able to handle arbitrary geometries in a consistent manner. These two techniques share the same issues of implementation on multicomputer architectures, though they differ somewhat in detail. The applicability of each method depends upon the feature of interest in a particular scattering problem. The Finite Difference Time Domain (FDTD) code is particularly suited to transient analysis for pulsed incident radiation. The Finite Element Method (FEM) is better at handling arbitrary scatterer geometry and composition.

4.2 Multicomputer Architectures

A multicomputer [2,3] may be thought of as a collection of n computers which communicate with each other over some kind of communications network. A generic multicomputer is illustrated in Fig. 4.2. Each processor (often referred to as a node) is in some sense a fully

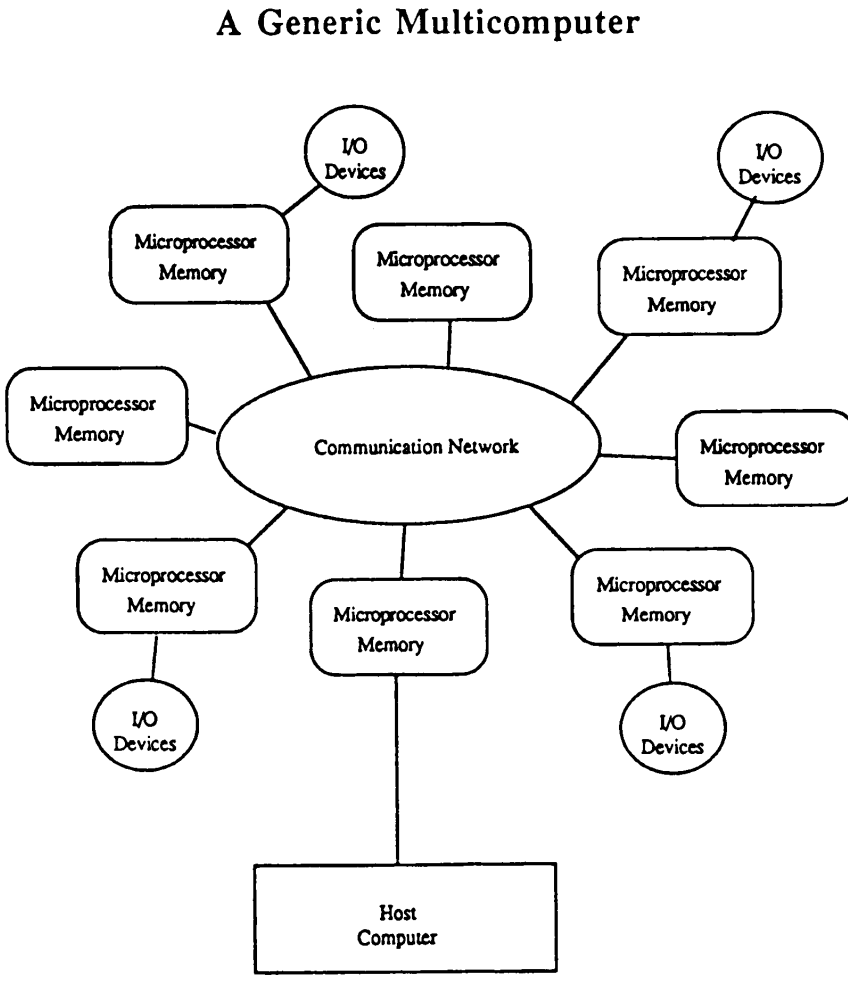

Figure 4.2 A generic multicomputer. A collection of N processors is bound together by a communication network. Each processor is an independent computer with local memory and possibly some I/O devices. Processors communicate with each other by sending messages over the communications network. The multicomputer typically requires another computer as its host.

functional computer, consisting of a microprocessor, some compliment of local memory, and perhaps some external I/O devices. Typical floating point performance of present day multicomputer microprocessors is in the range of 0.1 to 40 megaflops. The amount of memory attached to each processor tends to scale with the floating point performance, and ranges from 0.5 to 16 megabytes. However, no processor may directly access another's memory. Instead, processors communicate with each other and coordinate their activities by sending messages across the communication network. Since each is an independent computer, different processors may execute different instructions or even entirely different programs at the same time. Thus there is a great deal of flexibility in the computational strategy which can be used on this type of machine. To the extent that the programmer can keep all of the processors busy doing useful work, an n processor multicomputer executes computations n times faster than a single processor can.

The communication network can take many forms (see Fig. 4.3). The ideal network for a multicomputer is a fully connected topology, where each processor can talk directly to every other processor. Unfortunately, the number of connections required in a fully connected topology scales as n^2, which is not technically feasible for more than a handful of processors. Another popular topology is the hypercube connect topology. The multicomputer is designed with $n = 2^p$ processors. Each processor is directly connected to p other processors so that the network looks like a p-dimensional hypercube, with processors at the hypercube corners and communication links along the hypercube edges. The number of connections in this type of network scales as $n \log_2 n$, which is more manageable over a wide range of n. Messages must make multiple hops, however, in the majority of cases. The maximum number of hops required to reach the farthest processor is p. The Caltech/JPL Mark IIIfp is a hypercube connected multicomputer. Some other multicomputers use a periodic mesh topology. Here, each processor is connected to four other processors in a 2-dimensional periodic mesh. The number of connections scales as $4n$ for this case. Again, multiple hops are required to reach most other processors, with the maximum number of hops equal to $\sqrt{n}$ on a square mesh. The topology of the mesh can impact the performance of an algorithm because sending messages to distant processors takes longer than communicating with a neighbor. Proper mapping of a program onto the communications network is therefore a concern.

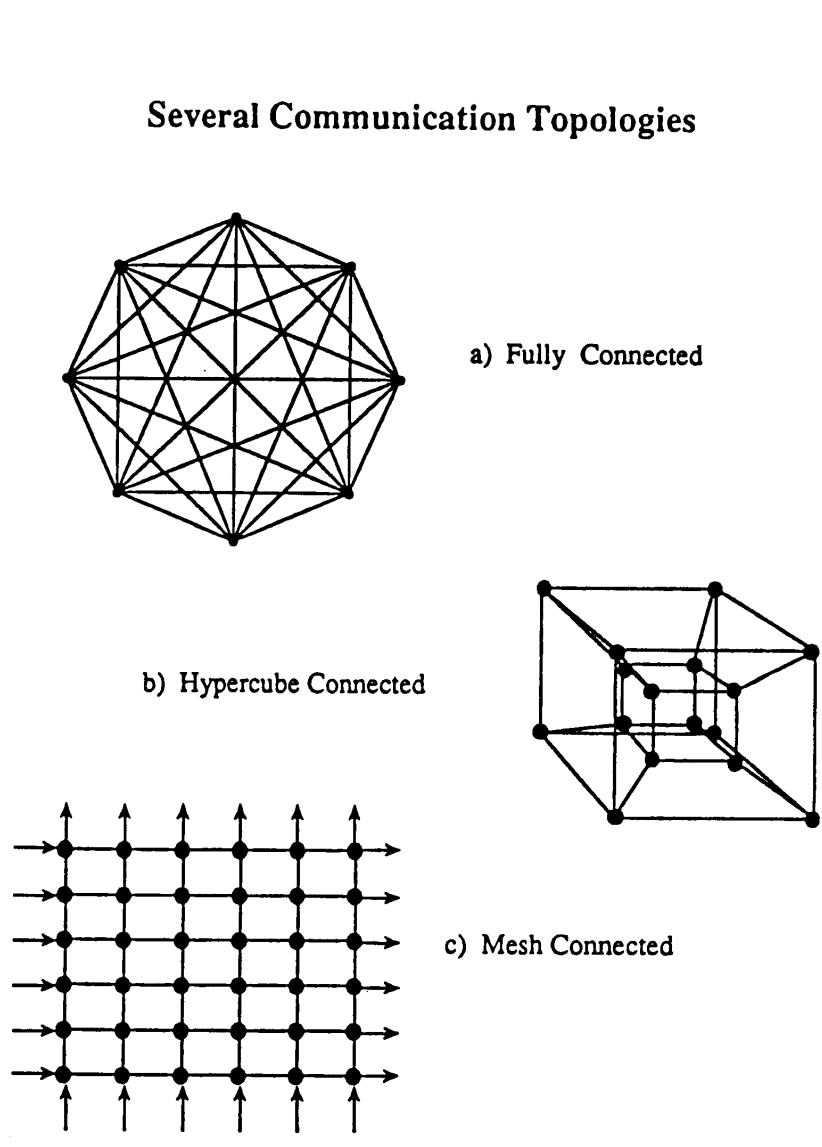

Figure 4.3 Several communication topologies for multicomputers. (a) Fully connected. Each processor has a link to every other processor. (b) Hypercube connected. 2^m processors are linked together as an m-dimensional hypercube. (c) Periodic mesh connected. Each processor is connected to four others on a rectangular communications mesh.

The Caltech/JPL Mark IIIfp Hypercube consists of up to 128 processors connected via bit serial communication channels in a hypercube topology. Each processor, or node, consists of a pair of Motorola 68020 microprocessors, a Motorola 68882 Floating Point Coprocessor, a Weitek processor daughter board, and four megabytes of main memory. One 68020 is the data processor, while the second 68020 is dedicated to communication control. The processors share the main memory. Each processor can be programmed separately, but the usual mode of operation is to run a system communication kernel on the communication processor and confine user programs to the data processor or Weitek processor. Some more advanced applications make use of both the data processor and the Weitek processors simultaneously, with the Weitek being treated as a subroutine engine (much like an attached array processor on sequential machines). We have chosen a simpler approach, where either the data processor or the Weitek is used, but not both. Our programs are written in FORTRAN 77, with communication between processors handled by interface subroutines to the communications processor. Communication between processors is explicitly coded in our programs. Although the operating system on the Mark IIIfp provides several types of communication primitives, our codes were build using only two of them: the nearest neighbor synchronous message exchange and the broadcast. More complicated functions, such as generalized message exchange and global data summation, were built from the nearest neighbor exchange function. All of our results were obtained with programs running on the Weitek processor, which typically gets about 1–2 megaflops performance from FORTRAN code. FORTRAN I/O is handled by CUBIX, a UNIX-like interface to the host, which allows data to be read or written by each processor severally or in unison.

There are two major considerations in designing algorithms for multicomputers. The first is the issue of load balance. Supercomputer processing power is obtained by having all processors doing computations concurrently. If some processors have less work to do than others, performance necessarily declines as processors become idle waiting for the rest to complete their tasks. An algorithm must therefore seek to maintain an even distribution of the work among all processors in order to obtain peak performance. The second issue is that of communication latency. A typical characteristic of all multicomputers is that the time required to do a floating point operation T_{fp} is small compared to the

time T_c required to communicate a floating point number from one processor to another. Additionally, any communication involves some overhead, so that a typical expression for T_c is

$$T_c = T_l + \alpha N_b \tag{1}$$

where T_l is the latency, α is the transmission rate, and N_b is the number of bytes in the message. On the Mark IIIfp Hypercube, T_l is about 100 microseconds and α is 5 megabits per second, while T_{fp} is about 0.5 microseconds. Thus an effective algorithm also seeks to minimize the ratio of communication to computation, and maximize the size of the messages involved. For many types of scientific computations, the cost of communication is the least important of the two. Load balance is the primary concern.

Two standard performance measures are used in evaluating codes running on multicomputers. The first is fixed problem size speedup. If T_1 is the execution time of a code for a given problem on one processor (which implies that no communication is required) and T_n is the execution time for the code for a given problem on n processors, then S_n, the n processor speedup is defined as

$$S_n = T_1/T_n \tag{2}$$

In an ideal implementation, a multicomputer code running on n processors should execute n times faster than on one processor, resulting in a speedup of $S_n = n$. In practice, some overhead is incurred due to interprocessor communication and processor load imbalance, with a resulting speedup which is less than n. A measure of how efficiently the processors are being utilized is defined by

$$E_n = S_n/n \tag{3}$$

Fixed problem speedup always has an upper bound, since at some point there will be more processors than the total operation count required to solve the problem. In practice, the upper bound is reached far sooner, because the additional communication overhead incurred by adding new processors outweighs the reduction in computation.

The second performance measure can be thought of as algorithm scalability. If $T_1(P(1))$ is the time required to execute a code for problem size $P(1)$ on 1 processor, and $T_n(P(n))$ is the time required to

execute the same code for a comparable problem which is computationally n times as large on n processors, then we can define scalability as

$$A_{n,P} = T_n(P(n))/T_1(P(1)) \tag{4}$$

Here we measure how a code will perform as the computational size of the problem is scaled with the number of processors. The point of this measure is to determine how the communication requirements will scale with increasing machine size. How exactly to scale a problem with number of processors so that the per processor computation load remains constant presents several difficulties. Consider an algorithm which involves Gaussian elimination, which runs in order m^3 time, where m is the rank of the matrix. The time to fill a rank m matrix scales as m^2, as does the memory required for storage of the matrix. To maintain a constant computational load for the matrix fill, m must be scaled as the square root of the number of processors. For the Gaussian elimination, the matrix rank must be scaled as the cube root of the number of processors in order to keep the amount of computation done in each processor a constant. To keep the per processor computational load constant for the entire algorithm requires a problem scaling somewhere in between. Thus, for a complex code composed of several parts, each with different dependencies on problem size, it is rather difficult to construct a set of scaled problems which keep the computation load per processor fixed.

If the problem size can be varied with the number of processors so that the amount of computation (excluding communication related computation) done by each processor remains constant, then an algorithm with unitary (perfect) scaling will take the same amount of time to execute on n processors as it does on 1 processor. Any increase in execution time represents the cost of communication. Scaling problems in this manner can be difficult to do in practice. Therefore, we simply scale problems in any convenient fashion, and correct the run times by the theoretical dependency of the algorithm on problem size. This leads to the following alternate definition of algorithm scalability :

$$A_{n,P} = \frac{T_n(P_n))}{T_1(P_1)} \left[\frac{n}{f(P_n, P_1)} \right] \tag{5}$$

where

$$f(P_n, P_1) = T_1(P_n)/T_1(P_1) \tag{6}$$

is the problem size scaling function. We generally calculate $f(P_n, P_1)$ from theoretical considerations. (It is usually not possible to measure $T_1(P_n)$ directly, because of memory constraints, run time constraints, or both.) For example, consider again Gaussian elimination. If we scale the matrix rank with number of processors, then $f(P_n, P_1) = n^3$. If instead we try to keep memory usage constant by scaling the matrix rank by $\sqrt{n}$, then $f(P_n, P_1) = n^{3/2}$. For codes which have several parts with different problem size scalings, we use the scaling function for the code section which is the most limiting as problem size goes to infinity.

In designing an algorithm for multicomputers, several approaches are possible. For example, each processor may execute exactly the same program, but with different input data. Except for the initialization phase of the computation, where each processor is given its dataset to process, no interprocessor communication is required. This strategy, known as trivial parallelization, is the easiest to implement, and is particularly useful for parameter studies. It has the drawback of requiring that the entire computation must fit into a single processor.

A second strategy breaks the computation into a series of steps which must be completed in sequence for a large collection of data. Each step is assigned to one or more processors. As a processor completes its computation on a subset of the input data, it passes its results on to processors which are to handle the next step and begins again with a new subset of the input data. This strategy is know as pipelining. It is somewhat more difficult to implement, and is only applicable to a restricted class of computational problems. Maintaining load balance can also be a problem.

The strategy most often used in scientific computing is that of problem partitioning, which is sometimes referred to as "divide and conquer." Each processor executes substantially the same program, but on a piece of the problem. Processors remain loosely coupled throughout the computation, exchanging data when necessary. This strategy is necessary to solve the largest problems, where all available memory is required. Programming using this strategy tends to look the same as programming for a single-processor computer. The difficulties lie in deciding how to partition the problem so that load balance is maintained, and redesigning traditional algorithms to operate distributed among a collection of processors.

The partitioning strategy is the one used by both the FDTD and

FEM codes. The gridded problem domain is partitioned among the available processors so that each processor has a piece of the problem. For communication efficiency reasons, the pieces have some small overlap. Processing proceeds in a loosely synchronized fashion, with every processor doing approximately the same thing at any point in the calculation. All processors synchronize with each other when communication is required. In both cases, the amount of computation scales with the volume of the grid, while the amount of communication scales with the surface area of the partitions. This leads naturally to a partitioning strategy which tries to maximize the volume of each partition while minimizing its surface. How this is accomplished for each code is quite different, and depends upon the attributes of these two algorithms. We now describe each algorithm in detail, and analyze its performance on the Mark IIIfp Hypercube.

4.3 The Finite Difference Time Domain Method

We describe here a concurrent implementation of a Finite Difference Time Domain code for Maxwell's equations. The code has evolved from a version supplied by A. Taflove [4] with the finite difference algorithm essentially unchanged. We will not repeat the derivations here. Instead, we will briefly review the method and concentrate on the details of its concurrent implementation.

The code uses a leapfrog scheme in time and space to follow the evolution of electric and magnetic fields in a box. Faraday's Law and Ampere's Law, in the absence of free charges and currents, can be written as

$$\nabla \times \overline{E} = -\frac{\overline{\overline{\mu}}}{c} \cdot \frac{\partial \overline{H}}{\partial t} \tag{7}$$

$$\nabla \times \overline{H} = \frac{\overline{\overline{\epsilon}}}{c} \cdot \frac{\partial \overline{E}}{\partial t} + \frac{4\pi \overline{\overline{\sigma}}}{c} \cdot \overline{E} \tag{8}$$

where $\overline{\overline{\mu}}$ is the magnetic permeability tensor, $\overline{\overline{\epsilon}}$ is the electric permittivity tensor, and $\overline{\overline{\sigma}}$ is the conductivity tensor. We have assumed that the media constants are time independent, and that the linear relations $\overline{D} = \overline{\overline{\epsilon}} \cdot \overline{E}$ and $\overline{B} = \overline{\overline{\mu}} \cdot \overline{H}$ hold. Ferromagnetic and non-linear or time dependent materials require a more complicated treatment which we will not discuss here. We have also used Ohm's Law, $\overline{J} = \overline{\overline{\sigma}} \cdot \overline{E}$, to replace the conduction current in Ampere's Law.

The finite difference equations are obtained from the differential equations by using 2-point centered differences for space and time

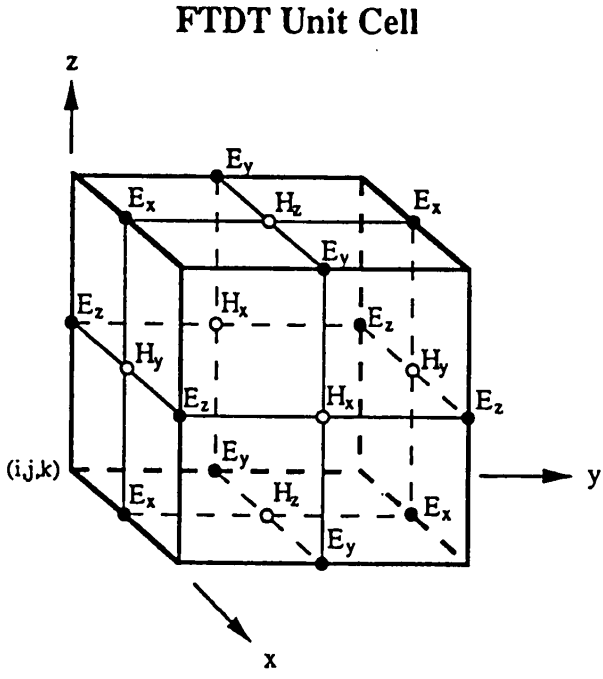

Figure 4.4 Locations of field grid points in an FDTD unit cell. Electric and magnetic field components are located on edges and faces where they are guaranteed to be continuous.

derivatives. The computational domain is tiled with a set of unit cartesian cells upon which the electric and magnetic field components are to be computed. Centered differencing is required to obtain a set of equations which are stable for iteration. To accomplish this, the electric field grid points are staggered in time and space with respect to the magnetic field grid points. A sample unit cell is illustrated in Fig. 4.4, with cell vertices located at integer cartesian coordinates. The electric field components are to be computed at the midpoints of the cell edges, while the magnetic field components are computed at the centers of the cell faces. Notice that only the component of the electric field parallel to that cell edge is computed at each mid-edge grid point. Likewise, only the component of the magnetic field perpendicular to the cell face is computed at each mid-face grid point. Permittivity and conductivity tensor values are assigned at each electric field location, while perme-

ability tensor values are assigned at each magnetic field location. These field component locations guarantee that centered spatial differences are taken in computing field updates. Time-centered derivatives are obtained by computing electric field values at integer time steps while magnetic field values are computed at half-integer time steps.

Consider the finite difference equation which corresponds to the x component of (7) (for simplicity, only diagonal tensor quantities are considered here):

$$\begin{aligned} -\frac{\mu_{xx}}{c}\frac{H_x^{n+\frac{1}{2}}(i,j+\frac{1}{2},k+\frac{1}{2}) - H_x^{n-\frac{1}{2}}(i,j+\frac{1}{2},k+\frac{1}{2})}{\Delta t} \\ = \frac{E_z^n(i,j+1,k+\frac{1}{2}) - E_z^n(i,j,k+\frac{1}{2})}{\Delta y} \\ - \frac{E_y^n(i,j+\frac{1}{2},k+1) - E_y^n(i,j+\frac{1}{2},k)}{\Delta z} \end{aligned} \tag{9}$$

The x component of $\overline{H}$ at the $n+\frac{1}{2}$ timestep can be computed from H_x at the $n-\frac{1}{2}$ timestep and the centered differences which correspond to the x component of the curl of $\overline{E}$ at the n timestep. The other components of $\overline{H}$ are updated in a similar fashion. The finite difference form of the x component of (8) is

$$\begin{aligned} &\frac{\epsilon_{xx}}{c}\frac{E_x^{n+1}(i+\frac{1}{2},j,k) - E_x^n(i+\frac{1}{2},j,k)}{\Delta t} + \\ &\frac{4\pi\sigma_{xx}}{c}\frac{E_x^{n+1}(i+\frac{1}{2},j,k) + E_x^n(i+\frac{1}{2},j,k)}{2} \\ &\qquad = \frac{H_z^{n+\frac{1}{2}}(i+\frac{1}{2},j+\frac{1}{2},k) - H_z^{n+\frac{1}{2}}(i+\frac{1}{2},j-\frac{1}{2},k)}{\Delta y} \\ &\qquad\quad - \frac{H_y^{n+\frac{1}{2}}(i+\frac{1}{2},j,k+\frac{1}{2}) - H_y^{n+\frac{1}{2}}(i+\frac{1}{2},j,k-\frac{1}{2})}{\Delta z} \end{aligned} \tag{10}$$

The x component of $\overline{E}$ at the $n+1$ timestep is computed from E_x at the n timestep and the centered differences which correspond to the x component of the curl of $\overline{H}$ at the $n+\frac{1}{2}$ timestep. In order to maintain time centering in the electric field update, the conduction term is computed as the average of the old and new $\overline{E}$ field values. The

spatial locations of the field components involved in (10) are depicted in Fig. 4.5. The other electric field components are similarly updated. The size of the computation cell must be chosen so that it is small compared to the fields' wavelength. The size of the time step is then determined from the Courant stability condition [5].

A boundary condition is required on each of the exterior faces of the computation box. We refer to these faces as truncation planes. For electromagnetic scattering computations, an outgoing wave boundary condition on the scattered field is required. Since the code is formulated in terms of the total field, a transition surface is defined inside the box which separates the box into two regions. The first is an interior region bounded by the transition surface which completely encloses the scatterer, and in which the total field formulation is applied. The second is an exterior region which lies between the transition surface and the truncation planes. The thickness of the exterior region is arbitrarily set to three cells. At the transition surface, the incident field is subtracted from the total field leaving only the scattered field to propagate in the exterior region. The outgoing wave boundary condition can be applied directly to the field components on the truncation planes. Only the electric field components on the truncation planes require a boundary condition, since the magnetic field components on these planes can always be computed from the electric fields there. The boundary conditions used in the code are taken from Mur [6].

A complete iteration on the field grid consists of first updating the magnetic field using the existing electric field, then updating the electric field using the new magnetic field, the boundary conditions, and the incident wave field on the transition surfaces. The typical initial condition for all field values is zero. Iteration on the field grid continues until a steady state is reached, or some predetermined timestep is reached. Steady state is determined by monitoring the magnitude and phase of the scattered field on an integration surface within the exterior region. Once steady state is reached, far fields can be obtained for radar cross-section calculations by doing a Green's function integration of the near fields on this surface.

4.4 Strategy for Concurrent Implementation

The computations performed in the main iteration loop of the FDTD code fall into five categories: magnetic field updates, boundary electric field updates, transition surface electric field updates, integra-

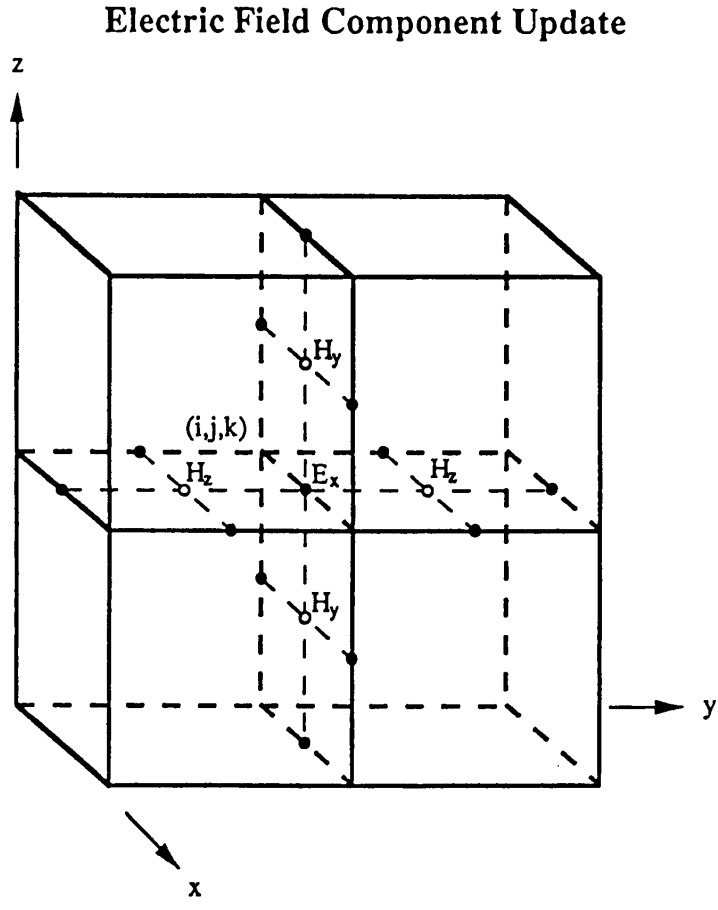

Figure 4.5 Magnetic field components involved in an E_x field update. Field components from several unit cells participate so that finite differences will be centered on E_x.

tion surface electric field updates, and all other electric field updates. Each of these categories has a different operation count per grid point. Since we intend to partition the grid among all processors, the effi-

ciency which the code can attain will depend upon how well the computational load from these various categories can be balanced with the communication which will be required when updating some grid points. Partitioning to obtain ideal load balance is a difficult problem. Fortunately, an intuitive decomposition gives very good results, provided that the number of cells in the computational box is large compared to the number of processors.

The decomposition used is essentially a bisection algorithm which proceeds as illustrated in Fig. 4.6. The computation box is split at the cell level into two roughly equal parts along its longest edge. These two parts are then split along their longest edge. The splitting of subboxes continues until the number of boxes equals the number of processors. The grid points belonging to each cell are kept together. The subboxes are assigned to processors so that subboxes which are adjacent in the physical space are in adjacent processors. This is always possible on a hypercube communication topology. Any interprocessor communication required can be done using direct channel connections without multiple processor hops. For 2D mesh communication topologies, restricting the partitioning to two of the three physical dimensions of the computation box may be more efficient, since a three-dimensional decomposition cannot in general be mapped onto a two dimensional mesh of processors without requiring multiple communication hops. (The advent of hardware point-to-point message routing may eventually nullify this concern. Such hardware establishes a direct communication path from one processor to another before messages are sent, eliminating the need to "store and forward" messages.)

No attempt is made in this decomposition to balance the computational load of each of the five categories of grid points mentioned previously. As a result, each processor will have differing numbers of grids in each category, and a different amount of computation overall at each iteration step. Fortunately, the number of magnetic field grids and normal electric field grids far outnumber the other three grid types, so the extra computation associated with the three special grid categories is a small perturbation on the total computation per time step. And so long as the number of cells assigned to each processor is large, the relative variation in the computational load of each processor will be inconsequential.

For most of the grid points in a subbox, the field update at each time step can be done using field information which is local to the

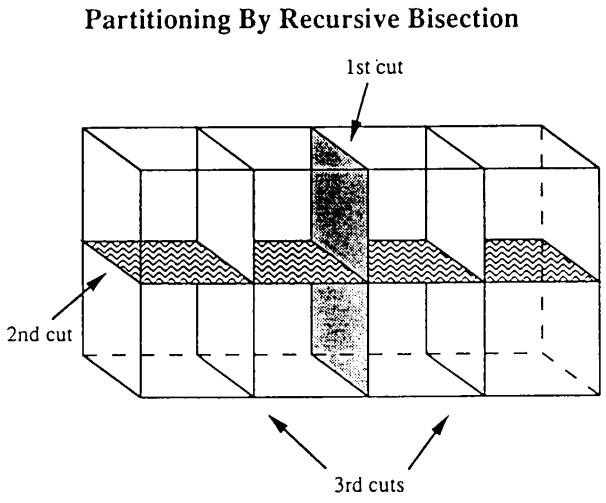

Figure 4.6 Partitioning an FDTD computational box by recursive bisection. The first cut is made perpendicular to the longest side, partitioning the FDTD grid cells (not shown) into 2 subboxes. Partitioning continues on the subboxes until the number of subboxes equals the number of processors.

processor. However, there will be a layer of electric and magnetic field grids which require field information from a neighboring processor in order to complete the update. The partition boundary and the grid points and cells adjacent to that boundary are illustrated in Fig. 4.7. Consider the update of the marked electric and magnetic field grids. For processor 2 to update E_x, it requires the value of H_z contained in processor 1. For processor 1 to update the value of H_z, it requires the value of E_x contained in processor 2. Where partition boundaries meet, information from more than one processor will be required for some grid point updates. To communicate this information on a grid-by-grid basis would be grossly inefficient, so the required grid points are duplicated in neighboring processors. We refer to these duplicate grid points as guard grids. The main iteration loop of the FDTD code can then proceed as follows:

(1) update all local magnetic field grid points (but not magnetic field guard grids) using local electric field grid information plus the electric field guard grids.

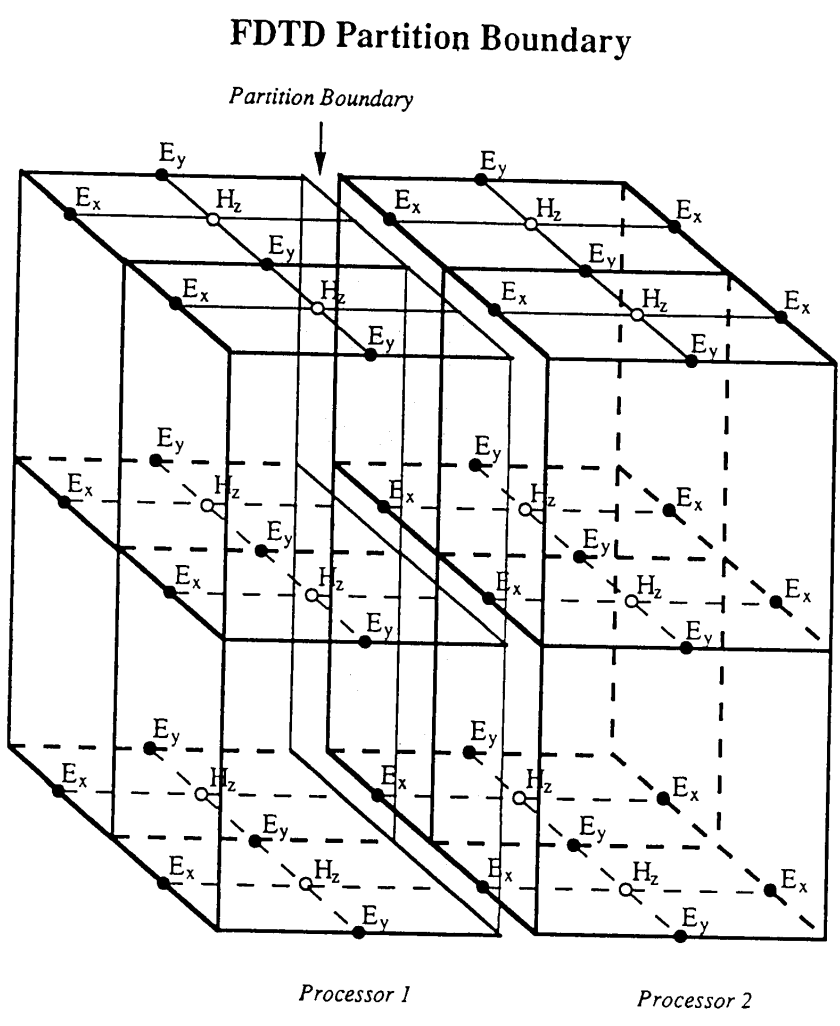

Figure 4.7 Some field grid points and cells on either side of a partition boundary. To update the E_x field grids adjacent to the partition boundary in processor 2, the values of the H_z field grids in processor 1 are required. Likewise, the update of H_z field grids in processor 1 requires the E_x field grids from processor 2. Not all field grids in each cell have been depicted.

(2) communicate all updated magnetic field values which correspond to magnetic field guard grids elsewhere to the appropriate processor.
(3) update all local electric field grid points (but not electric field guard grids) using local magnetic field grid information plus the magnetic field guard grids.
(4) communicate all updated electric field values which correspond to electric field guard grids elsewhere to the appropriate processor.

Through the use of guard grids, all of the computation required to update each category of grid point can be done with locally available information, while consolidating and minimizing the amount of interprocessor communication required.

This decomposition is sufficient for efficient operation for the bulk of the calculations performed in the FDTD code. The remaining calculation, the RCS computation, is not particularly load balanced by this decomposition. Since it represents a small fraction of the time spent on an FDTD run, no special effort has been made to make it run efficiently. Those processors which do not have any grid points involved in the RCS calculation simply do nothing, resulting in a drop in efficiency for this section of the code.

4.5 FDTD Performance on the Mark III Hypercube

For the purpose of performance evaluation, we will concentrate only on the main iteration loop of the FDTD code. It consists of three types of operations: field quantities update, boundary condition enforcement, and magnitude and phase tracking. The field quantities update consists of advancing the electric and magnetic fields one time step, including exchanging the new field values with neighboring processors. The enforcement of an outgoing boundary condition on the electric field at the edges of the computation box takes place immediately following the electric field update, but before field values are exchanged. After the magnetic field update and exchange, magnitude and phase tracking is performed on the fields at the integration planes to determine if steady state has been attained. The field values are monitored for extrema (magnitude), as well as the time of peak field relative to a reference time (phase). Peaks in the field at each grid point are determined by computing field time derivatives at successive time steps.

The first test problem consisted of a $32 \times 32 \times 24$ cell grid in which a $10 \times 10 \times 10$ dielectric cube was imbedded. The problem size was chosen to be close to maximum size for a single processor while requiring each dimension of the box to be divisible by the largest power of two possible (to maintain load balance when partitioned). The details of the scatterer have no impact upon the performance of the loop. This problem was run on 1, 2, 4, 8, 16, 32, and 64 processor Mark IIIfp Hypercubes, with speedup computed from times measured during each run.

In Fig. 4.8 we have plotted fixed problem speedup (2) for the three sections of the code main loop, as well as for the entire main loop itself and the RCS calculation. The field update exhibits good efficiency over the range of machines, with a speedup of 48 on the 64 processor hypercube, because this code section maintains processor load balance and a high ratio of computation to communication throughout. On the largest hypercube, there are still 384 cells per processor.

The magnitude and phase computation manages to remain above 50% efficiency throughout, though it is clear that its speedup is beginning to saturate. Load balance is responsible for the performance decline with increasing machine size, since the magnitude and phase computation is performed only on the integration planes and essentially no communication is required. These planes form a box six cells smaller in each dimension than the computation box. The partitioning algorithm divides the area of these planes evenly among up to eight processors, where each receives a corner section. After that, eight processors will have corner sections, while the rest will have edge sections or faces, or possibly no piece of the integration planes at all. (In the 64 processor decomposition, eight processors have subboxes in the middle of the computation box which are entirely inside of the integration planes. These processors therefore have nothing to do during the magnitude and phase computation, and the boundary condition computation.)

The boundary condition section saturates quickly, with little speedup gain past eight processors. It suffers from the load balance problem exhibited by the magnitude and phase calculation, and communication overhead which dominates this code section almost from the beginning. The poor efficiency of this section drags down the overall efficiency of the code, as seen in the total main loop speedup. Even though the field update constitutes 77% of the computation time when

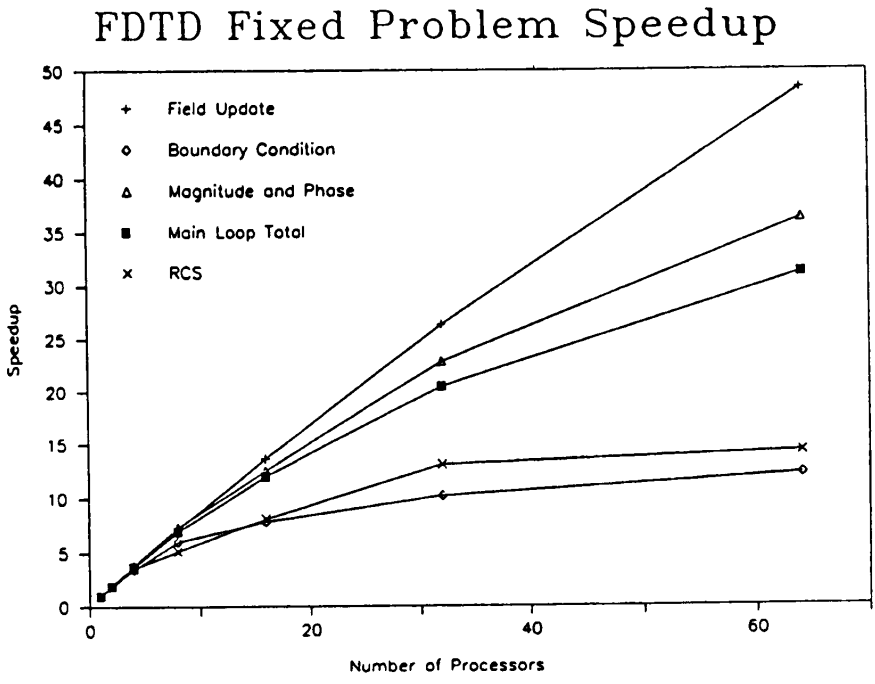

Figure 4.8 Speedup versus number of processors for several FDTD code sections. The problem size remained fixed for all cases. The FDTD main loop total is composed of the field update, boundary condition enforcement, and the magnitude and phase tracking. The RCS calculation is performed after the completion of all main loop iterations.

running on a single processor, its relatively good speedup compared to the other code sections brings it down to only 50% of the computation time on 64 processors. The boundary condition enforcement rises from 16% of the loop time to 41% of the loop time. Any further improvement in the efficiency of the entire loop will need to come from this code section.

For an estimate of performance on machine-limited problems we look at algorithm scaling measurements. In this test the problems size was scaled linearly with the number of processors so that the subbox which resulted from partitioning remained the same size throughout. This resulted in each processor always having a subbox of $32 \times 25 \times 20$ cells. The subbox size was chosen based upon the maximum problem

size for 64 processors. This subbox is smaller than the fixed problem due to increased memory requirements for communication. Again, times for each code section were measured on a range of hypercubes up to 64 processors.

Algorithm scaling as defined by (4) is plotted versus number of processors for each main loop section in Fig. 4.9. This is a problem where the computation load varies linearly (to first order) with the number of cells, so that (5) reduces directly to (4). The overall scaling of the main loop is excellent, as is the scaling of the field update section. The initial rise in the field update curve reflects the increase in communication caused by partitioning in coordinates orthogonal to the previous partition(s). After eight processors, the maximum number of neighbors any processor must communicate with remains fixed at six regardless of the number of additional partitions, resulting in a communications overhead which is independent of the number of processors. In fact, this scaling trend of an initial variation with number of processors which saturates and becomes independent of the number of processors is apparent in all sections of the main loop. This is directly related to the code's use of only nearest neighbor communications in each of the three main sections. The boundary condition and Magnitude and Phase sections decrease initially because the amount of computation being performed per processor is decreasing. On one processor, the entire outer surface of the box is involved in the boundary condition (and a related integration plane surface is involved in the Magnitude and Phase calculation). On two processors, one face of this surface has become a partition boundary which is interior to the problem, and no longer part of the boundary condition calculation. With four and eight processors, additional faces become partition boundaries. After eight processors, no further reduction occurs in the amount of outer boundary handled by the processors responsible for the corners of the computation box. The time spent on the boundary condition remains fixed at this level thereafter.

The scaling of the main loop reflects this decrease in the per processor computation load related to boundary surfaces. It also points out the difficulty in scaling problems so that the per processor computation load remains constant. But it is clear from the curve for the main loop of the FDTD code that by eight processors the communication overhead has saturated. A problem solved using the FDTD technique can be scaled with the number of processors to arbitrary size without

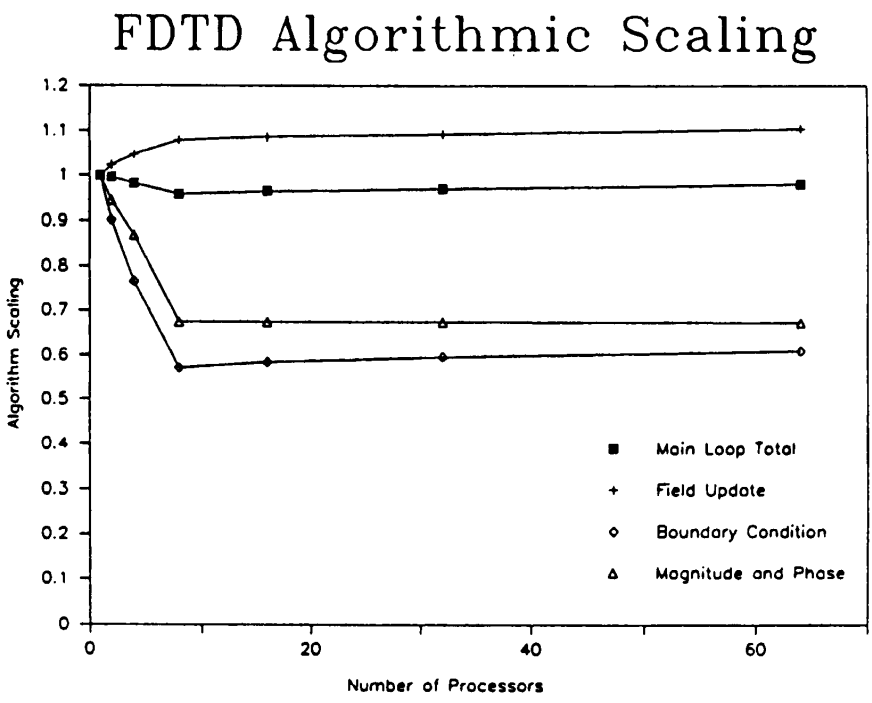

Figure 4.9 Algorithm scaling versus number of processors for several FDTD code sections. The problem size was scaled linearly with the number of processors. The field update, boundary condition enforcement, and magnitude and phase tracking constitute the main loop total. The flatness in all curves past eight processors indicates no growth in communication overhead.

having any detrimental impact on the code's performance.

One area which can have substantial impact on code performance is that of I/O. The FDTD code requires relatively little data on input and normally returns only RCS values for a set of angles distributed around the scatterer. The RCS calculation itself scales exactly like the boundary condition section of the code, since it operates on field data from the outer boundary. The relatively small amount of RCS data written out by the FDTD code has no impact upon the overall performance of the code. The code remains compute bound. If there were a need to write out field values for the entire grid, performance

of the code would be adversely impacted. We will see this effect in the Finite Element Method code, where the I/O requirements are quite different from FDTD.

4.6 The Finite Element Method

The finite element method (FEM) has been in use for many years in structural mechanics [7] and has become popular in recent years as a technique for use on electromagnetic problems [8]. FEM has the advantage of being able to deal with the specific geometry of objects by using unstructured gridding which follows an object's shape. This can be of particular importance in EM scattering problems, where the correct representation of a scatterer's surface is necessary for accurate computation. The theory of finite elements has been discussed in many places, so we will only summarize it as applied to electromagnetic problems. Rather, we will concentrate on the concurrent implementation of a finite element code which was written to solve the canonical EM scattering problem.

The code we describe solves scattering problems in the frequency domain. Beginning with Maxwell's curl equations, (7) & (8), for time independent media, we Fourier transform in time and combine the two to produce wave equations for either $\overline{E}$ or $\overline{H}$:

$$\nabla \times (\overline{\overline{\mu}}^{-1} \cdot \nabla \times \overline{E}) - \frac{\omega^2}{c^2} \overline{\overline{\epsilon}} \cdot \overline{E} = 0 \tag{11}$$

$$\nabla \times (\overline{\overline{\epsilon}}^{-1} \cdot \nabla \times \overline{H}) - \frac{\omega^2}{c^2} \overline{\overline{\mu}} \cdot \overline{H} = 0 \tag{12}$$

Either equation with appropriate boundary conditions constitutes a complete description of the scattering problem. We have dropped an explicit reference to conductivity, since in the frequency domain, conductivity and permittivity can be combined into a general complex $\overline{\overline{\epsilon}}$.

To apply the finite element method, we transform the differential wave equation into an equivalent "weak form" integral equation. Using (11), we multiply by a vector test function $\overline{T}$ and integrate over the problem domain. An integration by parts gives the weak form integral equation

$$\int_\Omega d^3v \left[\nabla \times \overline{T} \cdot \overline{\overline{\mu}}^{-1} \cdot \nabla \times \overline{E} - \frac{\omega^2}{c^2} \overline{T} \cdot \overline{\overline{\epsilon}} \cdot \overline{E} \right] = \int_\Gamma dA \, \hat{\overline{n}} \cdot \overline{T} \times (\overline{\overline{\mu}}^{-1} \cdot \nabla \times \overline{E}) \tag{13}$$

where Ω is the problem domain, Γ is the surface which bounds the problem domain, and $\hat{n}$ is a unit normal to the surface pointing outward. It can be shown that for a given boundary condition on $\overline{E}$ at Γ, if $\overline{E}$ satisfies this equation for an arbitrary test function $\overline{T}$ (which is subject to certain conditions of smoothness and value on Γ), then $\overline{E}$ also satisfies the differential equation from which (13) was derived. (An entirely equivalent integral equation can be obtained for $\overline{H}$). The computation domain Ω may now be decomposed into a set of finite elements.

For some problems, the full complexity of a three-dimensional description is not necessary. A two-dimensional computation may be entirely adequate. For these problems, a simpler integral equation may be derived. If we take z to be the invariant direction, then we may derive a Helmholtz equation for E_z from (11) & (12) which is a scalar equation:

$$\nabla \cdot \left(\frac{\nabla E_z}{\mu} \right) + \frac{\omega^2}{c^2} \epsilon E_z = 0 \tag{14}$$

The media constants ϵ and μ have been taken to be scalars. A similar equation may be obtained for H_z. The integral equation appropriate to the finite element method is obtained as above by multiplying by a test function and integrating over the problem domain.

$$\int_{\Omega} d^2v \left[\frac{\nabla T \cdot \nabla E_z}{\mu} - \frac{\omega^2}{c^2} \epsilon T E_z \right] = \int_{\Gamma} ds \, \frac{T}{\mu} \frac{\partial E_z}{\partial n} \tag{15}$$

Here $\partial E_z / \partial n$ is the normal derivative of E_z on the boundary. The boundary conditions of the problem are incorporated into the integral over the surface Γ (Neumann) or into the values of the field on the surface Γ.

As is the case with FDTD, finite elements require a finite computational domain, with some kind of outgoing wave boundary condition. For the 2D scalar case, we use the Bayliss-Turkel 2nd order absorbing boundary condition [9] on a circular outer boundary. This is a condition on the scattered portion of the field:

$$\left(\frac{\partial}{\partial (kr)} - i + \frac{5/2}{kr} \right) \left(\frac{\partial}{\partial (kr)} - i + \frac{1/2}{kr} \right) E^{scat} = 0 \tag{16}$$

We may eliminate the second derivative in r in favor of a second derivative in ϕ, the cylindrical azimuthal angle, by using (14) (which

the scattered field must satisfy in the region exterior to all scatterers):

$$\frac{\partial E^{scat}}{\partial r} = \alpha(r)E^{scat} + \beta(r)\frac{\partial^2 E^{scat}}{\partial \phi^2} \tag{17}$$

where

$$\alpha(r) = \frac{k}{kr - i}\left(-ikr - \frac{3}{2} + \frac{3i}{8kr}\right) \tag{18}$$

and

$$\beta(r) = \frac{-i}{2r(kr - i)} \tag{19}$$

We now have two choices. We can convert (15) into an equation for the scattered field, and incorporate the Bayliss-Turkel boundary condition into the boundary integral, or we can convert the Bayliss-Turkel boundary condition into a condition on the total field and apply the result directly to (15). In either case, we need to represent the total field in terms of the scattered and incident fields:

$$E^{tot} = E^{inc} + E^{scat} \tag{20}$$

Each formulation has different advantages. Incorporating (16) into (15) in the total field formulation gives the following weak form equation for E_z:

$$\int_\Omega d^2v \left[\frac{\nabla T \cdot \nabla E_z}{\mu} - \frac{\omega^2}{c^2}\epsilon T E_z\right] + \int_\Gamma d\phi \left[\beta(r)\frac{\partial T}{\partial \phi}\frac{\partial E_z}{\partial \phi} - \alpha(r)T E_z\right]$$
$$= \int_\Gamma d\phi \left[\beta(r)\frac{\partial T}{\partial \phi}\frac{\partial E_z^{inc}}{\partial \phi} - \alpha(r)T E_z^{inc} + T\frac{\partial E_z^{inc}}{\partial r}\right] \tag{21}$$

Here we have assumed that the outer boundary Γ is circular and lies in the vacuum region ($\epsilon = \mu = 1$). We have also done an integration by parts to eliminate the second derivative in ϕ. The scattered field formulation is easily obtained from (21) by substituting (20) and gathering terms in E^{inc} to the right-hand side. The resulting integral equation

$$\int_\Omega d^2v \left[\frac{\nabla T \cdot \nabla E_z^{scat}}{\mu} - \frac{\omega^2}{c^2}\epsilon T E_z^{scat}\right]$$
$$+ \int_\Gamma d\phi \left[\beta(r)\frac{\partial T}{\partial \phi}\frac{\partial E_z^{scat}}{\partial \phi} - \alpha(r)T E_z^{scat}\right]$$
$$= \int_\Gamma d\phi T\frac{\partial E_z^{inc}}{\partial r} - \int_\Omega d^2v \left[\frac{\nabla T \cdot \nabla E_z^{inc}}{\mu} - \frac{\omega^2}{c^2}\epsilon T E_z^{inc}\right] \tag{22}$$

has a left-hand side with the same form as (21). In theory, the two formulations should produce the same results. In practice, their accuracy differs, depending upon the problem, due to phase propagation errors.

The Bayliss-Turkel absorbing boundary condition is not directly applicable to the 3D vector fields problem. One can apply the Sommerfeld radiation condition to the boundary, provided that the boundary is far enough away from the scatterers. In 3D, pushing the outer boundary away from the scatterers increases the problem domain by the cube of the distance, a situation which is clearly intolerable and limiting. A useful local absorbing boundary condition in 3D is a topic of current research.

The 2D or 3D integral equation is transformed into a set of linear equations by decomposing the problem domain into a set of finite elements. Because it is easier to visualize, we will discuss the finite element method and its concurrent implementation in terms of the 2D scalar equation. Extension of the finite element method to 3D is straightforward, and is commonly in use in structural mechanics and fluid dynamics, though problems of field continuity, absorbing boundary conditions, and field divergence arise in electromagnetic problems. The problem domain Ω is meshed with nodal points at which the solution is to be found, matching the geometry of the objects. These nodes are then tiled with a set of finite elements (see Fig. 4.10). In 2D, the elements might be triangles or quadrilaterals, in 3D, tetrahedra, wedges, or bricks. A set of basis functions are defined at each node in the mesh, which have nonzero value only within the elements of which it is a part. These basis functions are generally some polynomial function which is 1 at the node defining it, 0 at all other nodes in the element, and 0 along the edges of the element opposite the defining node. An example of a linear basis function is given in Fig. 4.11. The function is continuous inside and across elements, dropping to zero at the element edges which do not intersect the node. On all other elements in the grid, the basis function is identically zero.

The field quantities (electric or magnetic) may be expressed as a linear combination of these basis functions :

$$E_z = \sum_i d_i \xi_i \tag{23}$$

where ξ_i is the basis function at the ith node, and d_i is its coefficient in the representation for E_z. Notice that since, by definition, all other

A Sample Finite Element Mesh

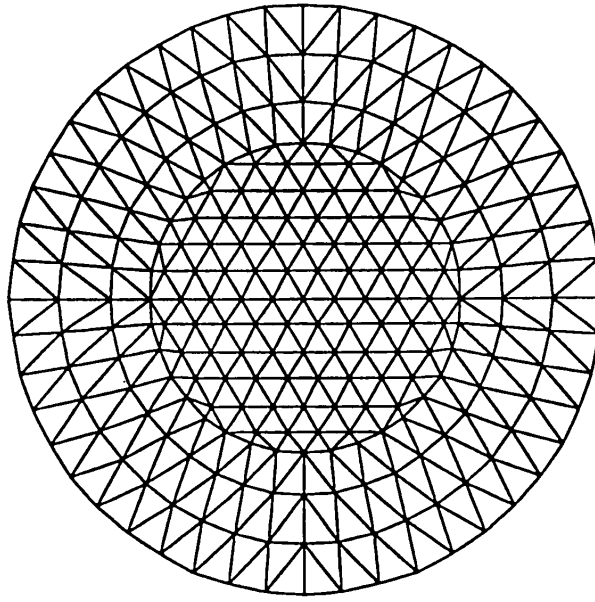

Figure 4.10 A sample finite element mesh. A circular domain Ω containing a circular region with different ϵ has been meshed with nodes and linear triangular finite elements. The nodal points are located at the vertices of the triangles. This mesh was constructed for a dielectric cylinder scattering problem.

basis functions are 0 at node i, the value of d_i is in fact the value of E_z at the ith node. We may also write the test function T in terms of these basis functions :

$$T = \sum_j b_j \xi_j \tag{24}$$

Substituting these into (21) results in the following matrix equation:

$$\overline{b} \cdot \overline{\overline{K}} \cdot \overline{d} = \overline{b} \cdot \overline{F} \tag{25}$$

where $\overline{b}$ and $\overline{d}$ are vectors composed of the coefficients of the basis function expansions in (23) & (24), and $\overline{\overline{K}}$ and $\overline{F}$, known as the

A Linear Nodal Basis Function

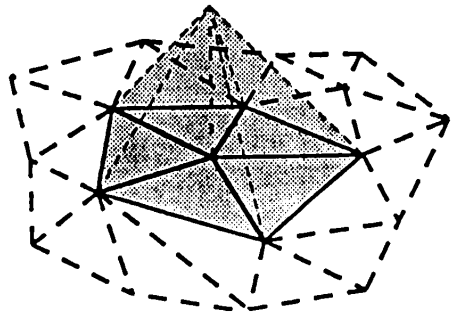

Figure 4.11 A linear basis function on a 2D mesh. The function is 1 at its parent node and is continuous across element edges. It falls linearly to 0 at all other nodes in the elements to which it belongs, and is identically 0 in all other elements.

stiffness matrix and force vector, are given by the expressions:

$$K_{ij} = \int_{\Omega} d^2v \left[\frac{\nabla\xi_i \cdot \nabla\xi_j}{\mu} - \frac{\omega^2}{c^2}\epsilon\xi_i\xi_j \right] + \int_{\Gamma} d\phi \left[\beta(r)\frac{\partial\xi_i}{\partial\phi}\frac{\partial\xi_j}{\partial\phi} - \alpha(r)\xi_i\xi_j \right] \tag{26a}$$

$$F_i = \int_{\Gamma} d\phi \left[\beta(r)\frac{\partial\xi_i}{\partial\phi}\frac{\partial E_z^{inc}}{\partial\phi} - \alpha(r)\xi_i E_z^{inc} + \xi_i\frac{\partial E_z^{inc}}{\partial r} \right] \tag{26b}$$

These integrals involve individual basis functions or products of basis functions. Since all basis functions are localized to a handful of finite elements, these integrals are non-zero only for those elements which contain the basis functions involved. This results in a $\overline{\overline{K}}$ matrix which is quite sparse. As a matter of practice, these integrals are computed on an element by element basis, with each element's contribution to $\overline{\overline{K}}$ and $\overline{F}$ added in its turn. In this manner, the complexity of integrating over a domain of irregular geometries is reduced to integrating over a set of regular finite sized elements.

Since E_z must satisfy (21) for any test function T, the vector of coefficients $\overline{d}$ of the basis function representation of E_z must satisfy (25) for any set of test function coefficients $\overline{b}$. This is equivalent to the requirement that $\overline{d}$ satisfy the set of linear equations given by

$$\overline{\overline{K}} \cdot \overline{d} = \overline{F} \tag{27}$$

Thus the solution of the integral equation has been reduced to the solution of a set of linear equations, which results in the field values at each node in the finite element grid.

This is the basic finite element method. A finite element code has two major computational sections, aside from the input of the geometry and output of the solution. The first section, which we call the element setup, consists of calculating the entries in the stiffness matrix $\overline{\overline{K}}$ and the force vector $\overline{F}$. This section typically has a loop which runs through the list of finite elements which make up the computational domain, adding each element's contribution to $\overline{\overline{K}}$ and $\overline{F}$ in turn. The second section solves the linear equations set given by (27), using some technique suited to solving sparse linear systems. The range of element types, methods of integration, issues of accuracy, considerations in 3D, and other technique specific issues are beyond the scope of this chapter. Instead, we now turn to the strategy used to implement such a code on concurrent computers.

4.7 Strategy for Concurrent Implementation

The bulk of the computation done in the finite element code occurs in the two code sections mentioned above: the element setup and the sparse linear equations solver. Efficient concurrent operation will require efficient implementations of each section. As was done for the concurrent version of the FDTD code, we will partition the computational domain among processors, seeking to maintain load balance and minimize interprocessor communication. But because of the irregular nature of the finite element grid, the domain decomposition algorithm will not be quite so straightforward. Again, we will confine our discussion to the 2D scalar FEM code for simplicity. The parallel implementation details for the 3D FEM code are unchanged.

The element setup section of the code computes 1 dimensional integrals involving 1 basis function along some edges of some elements, and 2 dimensional integrals involving 2 basis functions over the area of elements. For this code section, a decomposition like the one shown in Fig. 4.12 which distributes elements among the processors is required. So that all information required for the computation of the integrals in available locally, some nodal information will have to be duplicated in several processors. These nodes are referred to as shared nodes. For FEM grids composed entirely of one type of finite element, the number of 2D integrals per element is the same for all elements. Since

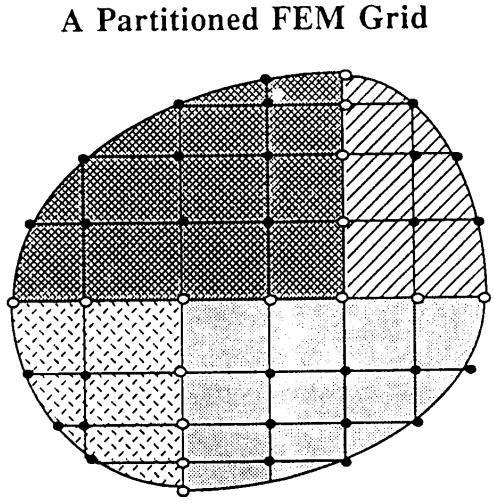

Figure 4.12 A partitioned finite element grid. Partitioning is done on an element basis. Some nodal points (black) are needed by only one processor, while others (white) must be shared by 2 or more processors.

the 1D integrals occur only on those elements which form part of the domain boundary, not every element contributes to them, and there are far fewer of them than there are 2D integrals. In this case we might ignore the extra computation associated with boundaries and simply divide the elements evenly among the available processors. If the FEM grid is heterogeneous in element type, then the number of 2D integrals per element will not be the same, making the determination of a load balanced decomposition more difficult. Fortunately, once the decomposition is completed, no additional communication is required to compute the integrals. Adding each integral's contribution into $\overline{\overline{K}}$ and $\overline{F}$ may require some communication at the end of the element setup section. The decomposition of $\overline{\overline{K}}$ and $\overline{F}$ among processors will depend upon the type of solver to be used.

The $\overline{\overline{K}}$ matrix which results from (26) has three major features. Its rank is large, especially in 3D, where small problems start at 1000 equations. It is symmetric when a symmetric boundary condition is used, though a statement of positive definiteness is not possible because $\overline{\overline{K}}$ is complex. And it is extremely sparse, due to the nature of finite element basis functions. Each equation will have only some

tens of non-zero terms, regardless of the rank of the matrix. An iterative solver which requires only the computation of dot products and matrix-vector products can compute these products in order n operations. Also, these operations can be done on partitioned matrices with little communication overhead. For these reasons, we have chosen to use a Bi-Conjugate Gradient solver.

The Bi-Conjugate Gradient (BCG) algorithm [10] for symmetric matrices is given in Fig. 4.13 with the modifications necessary for concurrent operation noted. The algorithm requires two basic operations: a matrix-vector product and a vector dot product. In a distributed-memory environment, the matrix and vectors will be partitioned so that these two operations will require communication among processors. The best partitioning scheme will depend upon the matrix structure. For our finite element generated matrices, the matrix structure comes directly from the finite element grid. An equation corresponding to a row of the stiffness matrix can be viewed as a coupling of one nodal point on the grid to all other nodal points with which it shares finite elements (see Fig. 4.12). The local nature of these equations suggests that a decomposition of the matrix and vectors which keeps elements together while minimizing the number of nodal points which must be shared will produce the most efficient concurrent implementation of this algorithm. The problem is how to determine such a decomposition.

The algorithm we use in partitioning a finite element mesh [11] is known as Recursive Inertial Partitioning (RIP). It does the partitioning by treating the mesh as a solid and making cuts based upon moments of inertia of that solid. The algorithm proceeds as follows:

(1) assign weights to each element based on the element type (which is an indication of the amount of computation associated with it).
(2) compute the center of mass of the grid, using the weights as each element's mass combined with its coordinate location.
(3) find the axis through the center of mass about which the moment of inertia of the grid is a minimum.
(4) partition the grid into two pieces with a plane perpendicular to that axis through the center of mass
(5) repeat steps 2–4 on the pieces until the number of pieces equals the number of processors.

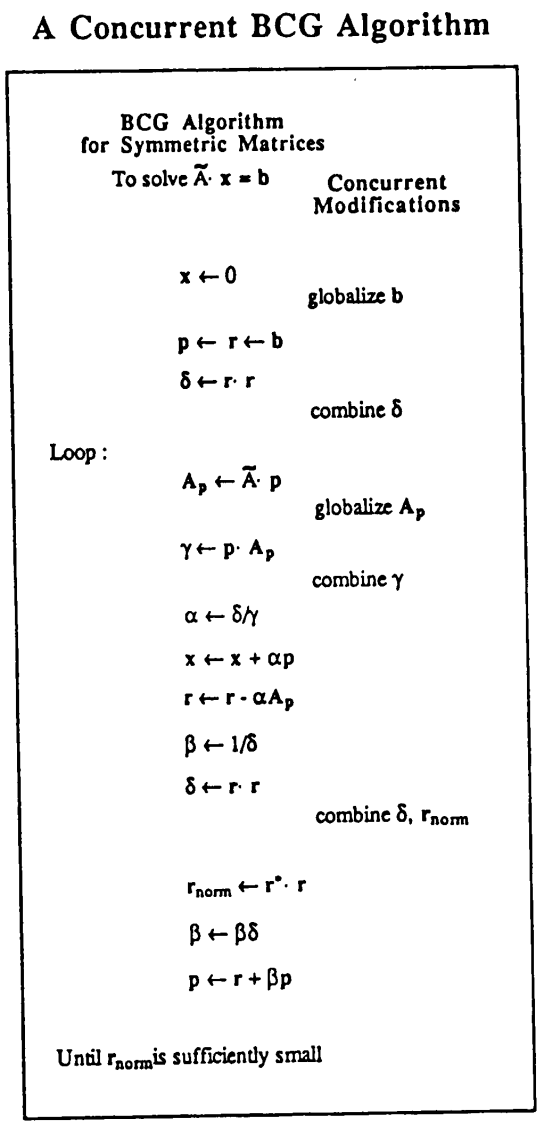

Figure 4.13 The bi-conjugate gradient algorithm with concurrent operation modifications. Distributed matrix-vector products require communication for computing shared node vector elements. Distributed vector dot products require summing together each processor's result.

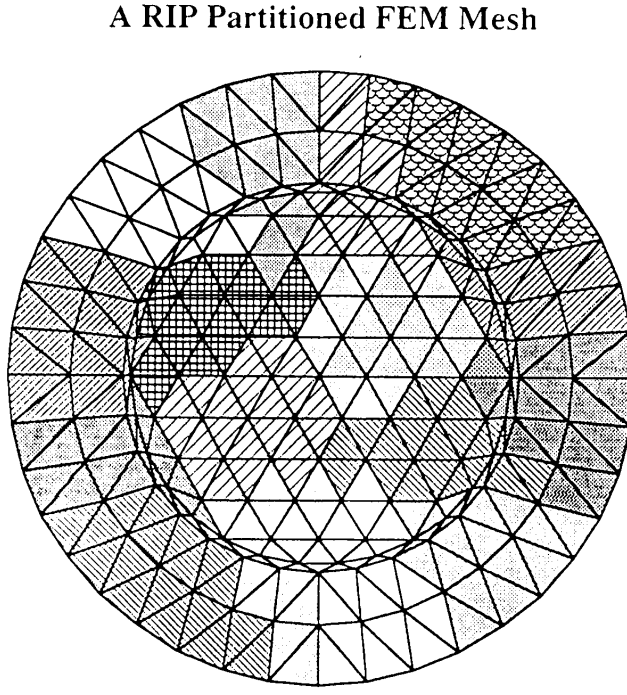

Figure 4.14 A RIP partitioned mesh. Elements are shaded by processor assignment. The algorithm attempts to produce compact partitions from the irregular grid. As the number of elements per processor becomes small, compactness becomes more difficult to maintain.

An example of a mesh which has been partitioned using RIP is given in Fig. 4.14. The algorithm produces pieces which tend to be compact, and have a minimal surface. Since the amount of data which must be shared (and communicated) among processors is proportional to the surface of these partitions, it is advantageous to make these shared surfaces as small as possible. Although the decompositions produced by RIP can be improved by hand-tuning, the efficiency gain which results is generally marginal.

Those nodes which fall on partition boundaries are designated as shared nodes, and are duplicated in the necessary processors. The rest of the nodes are designated internal nodes, and are assigned to exactly one processor. Non-zero entries in the $\overline{\overline{K}}$ matrix are computed and stored for all node combinations in a processor. Some entries which cor-

respond to shared nodes may have contributions from two or more processors. These contributions are not combined—they remain in their separate processors. Thus the $\overline{\overline{K}}$ matrix is never fully assembled. Force vector entries, as well as all other vector quantities, are assembled completely, with each processor having the entries which correspond to its internal and shared nodes. A distributed matrix—vector product is illustrated in Fig. 4.15. It is computed by having each processor compute a matrix—vector product with its portion of the stiffness matrix $\overline{\overline{K}}$, and the required vector. Then in a process we refer to as a *globalize* operation, processors add the partial product results which correspond to shared nodes with the partial results from other processors to get the correct result. To accomplish this, each processor sends messages containing its partial products to other processors who share nodes with it. For distributed dot products, it is important that contributions from shared nodes not be counted more than once. To avoid this problem, a single processor is designated as a node's owner by the partitioner. Partial dot products are then computed by each processor using only owned nodes, and the results are summed over all processors to get the complete dot product. This summation over processors is referred to as a *combine* operation. One globalize and two combine operations (see Fig. 4.13) are required per BCG iteration.

4.8 Performance of the FEM Code

Logically, the FEM code consists of four major sections. The first section is input/initialization, where the finite element model of the scatterers is read in along with parameters related to the incident field. In the concurrent version, partitioning information is also read in, so that each processor can decide which parts of the model it needs to keep. There is very little floating point computation associated with the input phase, although the finite element models themselves tend to be large. The next two sections are element setup and linear equations solver, which we have discussed above. The final section is field output, where the solution is written to a file suitable for additional processing.

We have measured speedup (2) for each of these code sections, as well as the entire code, and have plotted them in Fig. 4.16 as a function of number of processors. These measurements were made on a test problem of scattering from a dielectric cylinder with $\epsilon = 2.56$ and $ka = 1.0$. The finite element model contained 2304 quadrilateral elements (quadratic basis functions, 9 node elements) and 9313

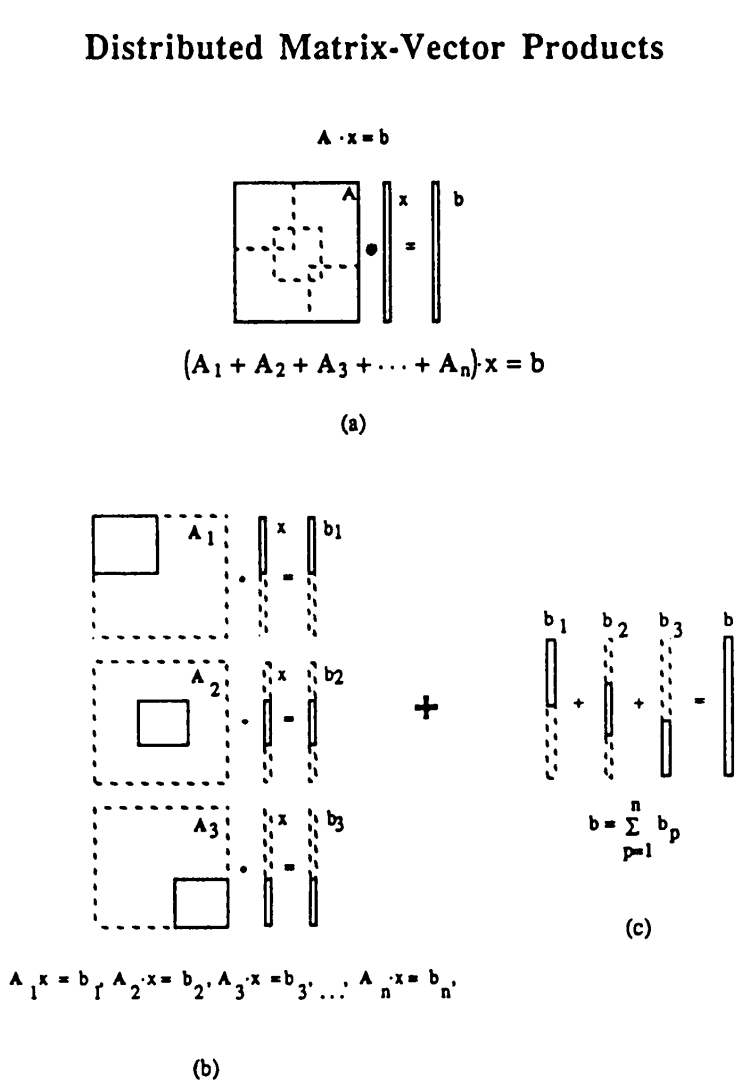

Figure 4.15 A distributed matrix-vector product. (a) The matrix $\overline{A}$ is partitioned into overlapping pieces. (b) Each processor can do a partial product using pieces of $\overline{A}$, $\overline{x}$, and $\overline{b}$. (c) The partial results must be summed over processors to get the correct result.

node points. This problem was chosen because it is close to the limit of problem size that can be done on one Mark IIIfp processor. The computationally intensive sections of the code exhibit almost linear speedup over the range of processors, with the setup exhibiting 95% efficiency and the solver achieving an efficiency of about 75% on 32 processors. Since there is no communication in the setup section, only load imbalance can impact performance there. From the graph it is apparent that the load balance is excellent. The solver starts out with high efficiency through eight processors, but begins to show communication overhead after that. (The load balance is known to be very good throughout.) The amount of communication increases as the number of processors increases, while the amount of computation decreases due to the decreasing size of each grid partition.

The two I/O sections of the code show poor performance, with the output section achieving a maximum speedup of 5, and the input section showing no speedup whatsoever. The limiting factor in the output section of the code is believed to be communication. Each processor is sending its field data to the host computer, where the data is written to disk. The hypercube is connected to the host through a single communication channel attached to processor 0, creating a bottleneck through which all data must flow. This bottleneck saturates with four processors. The input section of the code performs poorly because each processor is reading the entire input data set. This takes the same amount of time regardless of how many processors are reading it. As a result of the poor performance of the I/O related code sections, the overall performance of the FEM code is not good. It saturates at a speedup which is not much above 3.

Figure 4.17 demonstrates more clearly what has happened in this code. Here we have plotted the percentage of total execution time spent in each code section as a function of the number of processors. For the sequential version of the code (one processor), the I/O sections represent less than 35% of the execution time. On 32 processors, the I/O sections represent over 90% of the execution time. It is clear that adding additional processors will not improve the performance of the code on this particular problem any more, since it is now completely I/O bound. This is a potential problem with any code which requires large volumes of data to be input or output.

The speedup measurements presented here illustrate two important points about parallel processing. First, for a given particular prob-

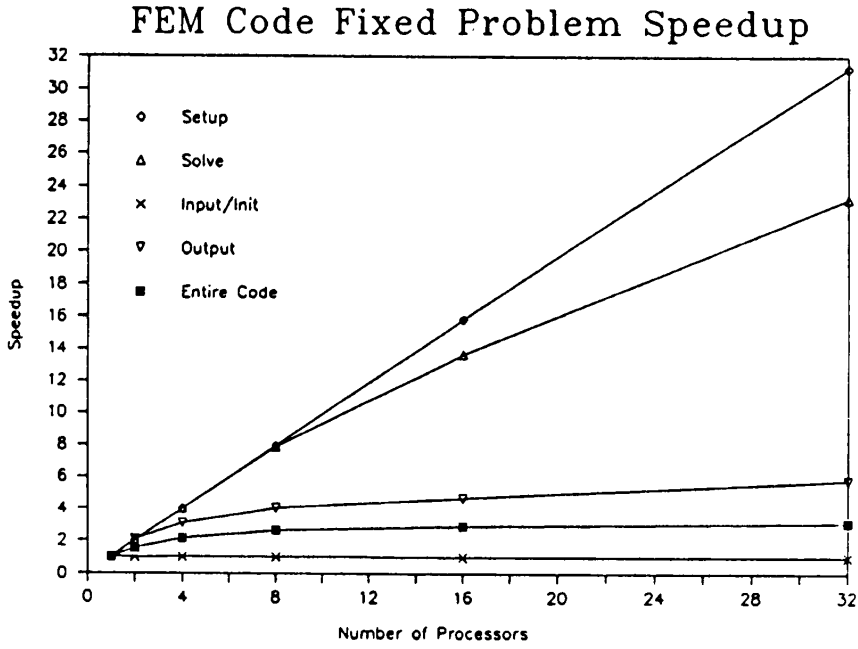

Figure 4.16 Speedup versus number of processors for several FEM codes sections. The problem size was the same for all runs. The Input/Init and Output code sections are dominated by file I/0. The Setup section is completely dominated by computation. The Solve section begins to show some communication overhead by the 32 processor case. The Entire Code suffers from poor performance by the I/O dominated sections.

lem, code performance will always saturate as the number of processors is increased. I/O tends to be more sensitive to this saturation because traditional approaches to reading and writing data on computers have a strong sequential aspect to them. Second, even though the most computationally intensive part of the code has been efficiently parallelized, what remains becomes the major performance inhibitor. It is true that a larger problem would saturate more slowly with increasing number of processors than did this example, but it would eventually saturate, and for exactly the same reasons. Thus speedup measurements, even on modest sized problems, are effective in pointing out where the hard work remains to be done.

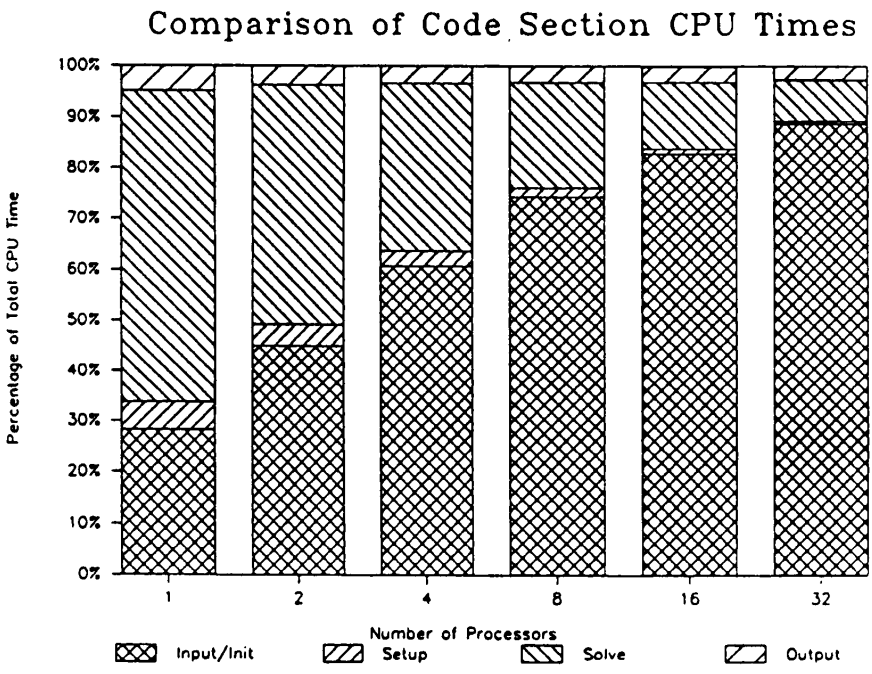

Figure 4.17 Code section percentages versus number of processors for the FEM code. On one processor, over 65% of the time is spent on computation. On 32 processors, over 85% of the time is spent on I/O. Uniformly good concurrent operation of all code sections would leave relative percentages unchanged.

To mitigate the I/O problems, we could employ a concurrent I/O facility for all data-related activity. This hardware allows many processors to read and write data simultaneously onto separate devices. In the limit where there is one device per processor, I/O should exhibit the same kind of speedups seen in the computational sections of the code. However, this is somewhat misleading in that input data from a source other than another code on the concurrent computer must be prepartitioned before it is useable, and the complement must be done for output data.

In terms of scaling of the algorithm to machine-limited problems, the outlook is in fact promising. The I/O time scales linearly with the size of the problem (as measured by the number of grid points

and elements in the mesh) while the solver should scale quadratically. Therefore, as the problems become large, the I/O time should become negligible compared to the time spent in the solver. Measuring the scaling of the concurrent Bi-Conjugate Gradient algorithm with number of processors is complicated by the fact that its performance depends in detail on the matrix. BCG can converge rapidly, or not at all, depending on the equations being solved. The reasons for non-convergence are poorly understood in terms of mesh descretization, geometry shape, etc. The execution time also depends upon the sparsity, and in its concurrent implementation, upon the compactness of the partitioning.

The FEM code scaling is difficult to analyze using (4), because of the different compute load dependencies on problem (mesh) size for the various code sections. Instead, we have constructed a series of test problems where the number of elements is directly proportional to the number of processors. These problems have rectangular finite element meshes of 9 node quadrilateral elements which, when partitioned by the RIP algorithm, result in each processor having a square mesh with 100 elements and 441 nodes. The scaling results for these problems are plotted in Fig. 4.18. We have assumed previously that the BCG solver run time will scale quadratically with the number of nodes in the mesh, so that $f(P_n, P_1) = n^2$ in (5). This is not exactly the case. We measured the solver scaling directly by counting floating point operations required for convergence in each run. We have plotted the scaling function $f(P_n, P_1)$ divided by n^2 in Fig. 4.18 as well. The curve shows that these problems do not present the solver with a set of tasks which scale uniformly with number of processors. We have therefore used this measured $f(P_n, P_1)$ in (5) for the remaining curves in Fig. 4.18.

The setup section rapidly becomes an unimportant part of the code, since its per processor cost remains fixed while the other sections are increasing with problem size. The I/O-dominated sections should also decline compared to the solver, but because of their poor parallel performance, they remain a significant part of the code execution time. The solver scaling shows a machine size dependence which is worse than the measured scaling function $f(P_n, P_1)$. The shape of the curve is logarithmic in the number of processors. This variation is due to communication overhead from the method used to exchange messages among processors when computing matrix-vector multiplies and

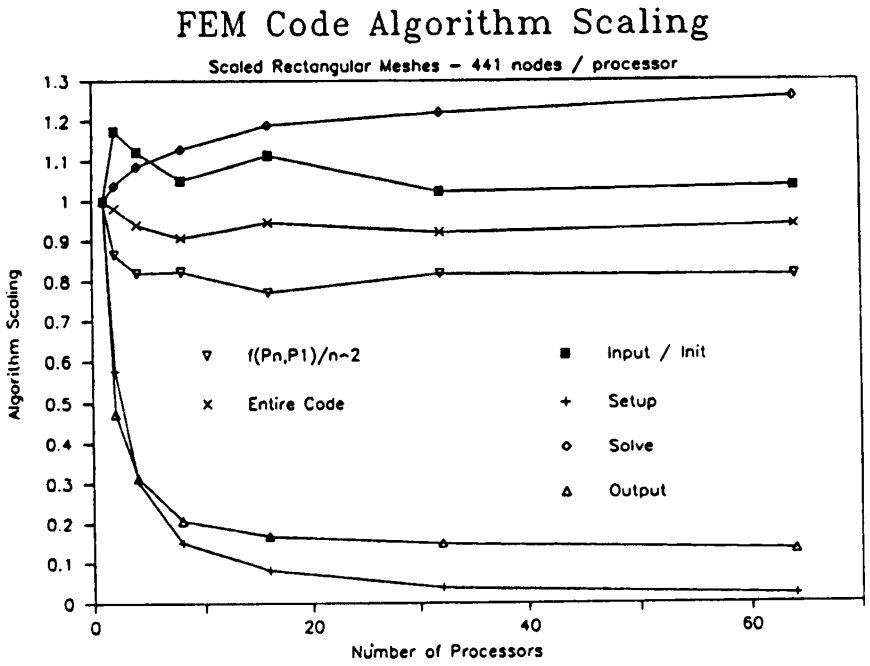

Figure 4.18 Algorithm scaling versus number of processors for the FEM code. The Solve code section will dominate for machine lmited problems on large numbers of processors. The logarithmic variation is an indication that communication overhead is being observed. The I/O bottleneck prevents the Input/Init and Output code sections from becoming insignificant.

dot products. In the FDTD code, the partitioning restricted communication to nearest neighbors. The RIP scheme cannot guarantee that partitions which are adjacent in physical space are assigned to adjacent processors, so the FEM code must use a general message passing scheme. On a hypercube connected machine like the Mark IIIfp, the overhead of a general message passing scheme increases logarithmically with the number of processors, which is the variation observed here. In the limit as the machine size goes to infinity, the communication overhead will eventually dominate everything else in the code. The logarithmic communications scaling is not readily apparent in the curve for the entire code because the I/O sections mask the effect. For

these test problems, the I/O sections represent approximately 45% of the execution time.

Fortunately, logarithmically increasing communication overhead requires very large numbers of processors before it dominates the run time on machine size-limited finite element problems. The poor I/O performance is still the major limiting factor in using this code on machines with large numbers of processors. There are several potential solutions to this problem. The obvious one is to improve the data transfer rate between the hypercube and the outside world. Another solution involves the amortization of the transfer of the FEM model to the hypercube over several executions of the code. In a production environment, the FEM code would be run for many incident angles on the same scatterer. The cost of transferring the model to concurrent I/O facilities needs to be paid only once. Accessing the finite element model for additional look angles can then be done in parallel. A third alternative would be to decrease the amount of input data by incorporate a finite element mesh generator into the code so that only a limited amount of geometry data needs to be read from the host. This is the strategy used by the FDTD code.

4.9 Conclusions

We have implemented two types of electromagnetic scattering codes on the Mark IIIfp coarse-grained multicomputer. Both types of codes solve Maxwell's partial differential equations on a grid for the near field surrounding the scattering objects. The implementations on the Mark IIIfp are expected to be representative of implementations on coarse-grained multicomputers in general.

The Finite Difference Time Domain (FDTD) code was tested on a variety of machine sizes (up to 64 processors) and found to run with high efficiency on all of them. The rectangular nature of the FDTD grid made distributing the problem among the processors a trivial task. The partitioning algorithm allowed nearest neighbor communication to be used for the most critical code sections, resulting in a code which remains compute bound for all machine sizes tested. Results of problem scaling tests indicate that the implementation will remain compute bound on machine-limited problems as the number of processors in the multicomputer is increased.

The Finite Element Method (FEM) code was written specifically for the Mark IIIfp Hypercube. A 2-dimensional scalar field version

was used for all of the performance evaluations reported here. A 3-dimensional vector field version is currently under development. Its performance characteristics are expected to be the same as its 2D predecessor. The computationally intensive sections of the FEM code were found to execute efficiently on a variety of machine sizes. Scaling measurements indicated that the communication overhead in the solver would become important as the number of processors in the multicomputer is increased, but that efficient execution could be expected on machine-limited problems over a much wider range of machine sizes than was tested.

The I/O sections of the code did not perform well on the multicomputer. The large volume of data required to specify the finite element model and the volume of field data returned from the code proved to be a severe limiting factor in the overall code performance. Since the code must wait for the entire model to be sequentially transferred to the hypercube before computation can proceed, and again for the field data to be returned sequentially to the host computer, the I/O sections quickly dominated the total computation time. This sequential bottleneck is perceived to be the most important problem with the FEM implementation on multicomputers.

The FDTD code does not routinely return field grid information. Rather, Monostatic and Bistatic RCS data is returned for a variety of look angles. Outputing field data from the code would have a detrimental effect on overall code performance due to the I/O bandwidth limitations on the Mark IIIfp Hypercube.

The computationally intensive sections of both codes were shown to perform efficiently on the Mark IIIfp and are expected to do so on other multicomputers. Compute bound performance is expected for both codes on machine size-limited problems. I/O involving the host computer (and with it the outside world) remains a problem requiring better hardware and innovative software restructuring.

Acknowledgments

Many people have had a hand in the work reported here. Ruel Calalo and Jean Patterson are primarily responsible for the initial conversion of Taflove's FDTD code to run on the hypercube. Jay Parker and Paulette Liewer have contributed to many areas in the develop-

ment of the finite element codes. Thomas Lockhart, Stephanie Mulligan, and Gregory Lyzenga are responsible for the implementation of the RIP algorithm and its interface to the finite element codes.

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

References

[1] Calalo, R. H., W. A. Imbriale, N. Jacobi, P. C. Liewer, T. G. Lockhart, G. A. Lyzenga, J. R. Lyons, F. Manshadi, and J. E. Patterson, "Hypercube matrix computation task, report for 1986–1988," *JPL Publication 88-31,* Jet Propulsion Laboratory, California Institute of Technology, Pasadena, CA, 1988

[2] Athas, W. C., and C. L. Seitz, "Multicomputers: message-passing concurrent computers," *IEEE Computer* **21,** 9–24, 1988.

[3] Fox, G. C., M. A. Johnson, G. A. Lyzenga, S. W. Otto, J. K. Salmon, and D. W. Walker, *Solving Problems on Concurrent Processors,* **1,** Prentice Hall, Englewood Cliffs, New Jersey, 1988.

[4] Umashankar, K. R., and A. Taflove, "Analytical models for electromagnetic scattering, part II: finite difference time domain developments," Final Report on IITRI Project E06538, Electronics Department, IIT Research Institute, Chicago, IL, June 1984.

[5] Press, W. H., B. P. Flannery, S. A. Teukolsky, and W. T. Vetterling, *Numerical Recipes,* Cambridge University Press, New York, 627, 1986.

[6] Mur, G., "Absorbing boundary conditions for the finite-difference approximation of the time-domain electromagnetic field equations," *IEEE Trans. on Electromagnetic Compatibility,* **EMC-23,** 377–382, 1981.

[7] Hughes, T. J. R., *The Finite Element Method,* Prentice Hall, Inc., Englewood Cliffs, New Jersey, 1987.

[8] Silvester, P. P., and R. L. Ferrari, *Finite Elements for Electrical Engineers,* Cambridge University Press, New York, 1983.

[9] Bayliss, A., M. Gunzburger, and E. Turkel, "Boundary conditions for the numerical solution of elliptic equations in exterior regions," *SIAM J. Appl. Math.*, **42**, 430, 1982.

[10] Jacobs, D. A. H., "The exploitation of sparsity of iterative methods," in *Sparse Matrices and Their Uses,* ed. I. S. Duff, Academic Press, London, 1981.

[11] Nour-Omid, B., A. Raefsky, and G. Lyzenga, "Solving finite element equations on concurrent computers," in *Proceedings of the Symposium on Parallel Computations and their Impact on Mechanics*, Boston, December 13–18, 1987.