1

COMPUTATIONAL ELECTROMAGNETICS AND SUPERCOMPUTER ARCHITECTURE

T. Cwik

1.1 Introduction

The tools used in electromagnetic engineering are derived from many branches of science and technology. The final design and implementation of an antenna component or low radar cross-section (RCS) object are typically the result of combining analytical methods, computational techniques, and experimentation. Arguably, computational techniques, as well as experimentation, have made the greatest progress over the last decade because they are both based on advances made in semiconductor technology, VLSI packaging, and computer architecture. These advances have been especially pronounced in the development of microprocessor power. Fig. 1.1 is a plot of the MFLOP performance over time needed to complete the LU decomposition of a matrix of order 1000, a common floating-point algorithm. The dramatic increase in performance over the last decade for microprocessor computations is compared with that for the supercomputer computations.

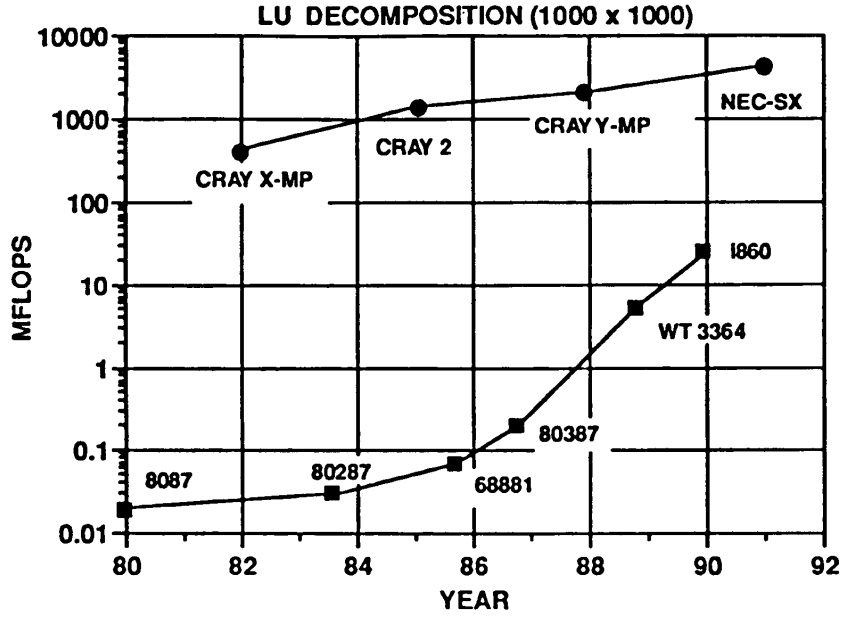

Figure 1.1 Evolution of floating-point performance for microprocessors and single processor supercomputers.

This performance, the projected performance, and a number of other issues such as cost and the inherent physical limitations in current supercomputer technology have naturally led to parallel supercomputers—an ensemble of interconnected microprocessors. Currently, hundreds to thousands of top-line microprocessors are interconnected to give overall performance rivaling and surpassing that of conventional supercomputers for a variety of physical simulations. Similarly, smaller numbers of the powerful vector processors are combined in parallel to achieve greater performance, but the ultimate number of these processors which can be combined is limited to far fewer than the number of microprocessors. In this chapter, an overview of computer architecture, as well as the application of parallel computer architecture to the solution of electromagnetic scattering problems, will be outlined. Because the majority of the material in this volume involves the use of parallel computers composed of many microprocessors, the review material in

this chapter will be mainly directed to those machines. Specifically, the decomposition, or 'breaking-up' of calculations so as to execute concurrently on a number of processors, will be presented for different electromagnetic techniques.

Initially, a short summary of relevant computer architecture is presented as background to the subsequent discussion. After the introduction of a programming model for problem decomposition, specific decompositions of finite difference time domain (FDTD), finite element, and integral equation solutions to Maxwell's equations are presented. The paper concludes with an outline of possible software-assisted decomposition methods and a summary.

1.2 Computer Architecture

Before examining the methods for mapping a given formulation of a scattering problem onto a parallel machine, it is necessary to examine different computer architectures from the perspective of the user. A much-referenced taxonomy of computer architectures was given by Flynn [1]. A hierarchical model of computer organization was presented in terms of instruction streams and data streams. This qualitative hierarchical model consisted of four categories:

(1) Single instruction stream, single data stream organization (SISD), which corresponds to the sequential execution of single instruction sets on single streams of data. Naively, conventional sequential computing falls into this category.

(2) Single instruction stream, multiple data stream (SIMD) organization, which corresponds to the identical instruction set simultaneously operating on multiple data sets, such as in an array processor.

(3) Multiple instruction stream, multiple data stream (MIMD) organization, which corresponds to different instruction sets operating on different data sets simultaneously. Many current parallel machines obviously fall into this category.

(4) Multiple instruction stream, single data stream (MISD) organization, which is a category that completes the hiearchary but does not have a useful implementation.

An organization based on instruction and data streams is but one

broad computer classification available. A second simplified classification is based on the number of processors and the distribution of memory. Fig. 1.2 shows a breakdown of machines into shared- and distributed-memory categories. A shared-memory machine consists of a small number, perhaps 1 to 64, of powerful processors, addressing a large bank of high-speed memory. Cray, NEC, and the Alliant companies, among others, make machines in this category. Distributed-memory machines consist of a larger number of processors and associated memory. Processors can be interconnected into various machine topologies—the hypercube and two-dimension mesh topologies are common. A hypercube topology consists of processors that are situated on the corners of an n-dimensional cube. This connectivity is optimal in the sense that there are only n cables connected to each processor and there is a maximum of n hops necessary to communicate data between any two processors. As communication hardware advances, this topology is giving way to mesh topologies. Distributed-memory machines are typically further divided into coarse- and fine-grained machines, depending on the number of processors and the amount of memory attached to each processor. Coarse-grained machines, which typically operate in MIMD mode, can have between 8 and 4000 processors, with each processor able to address 4 to 64 Mbytes of slower access memory. Memory addressing exists only between a processor and its attached memory, termed multiple addressing. For one processor to address another's memory, communication of the data between the processors is necessary. Intel and NCube companies, for example, build machines in this category. The fine-grained machines, which typically operate in SIMD mode, complete the spectrum, consisting of between 4000 and 128,000 simple processors addressing perhaps between 1 and 32 kbytes of slower access memory. Thinking Machines is the prime example of a company building machines in this category.

Though these models are useful in visualizing the organization of different machines, they are oversimplified because any one architecture can exhibit aspects of several categories. For example, Cray and NEC machines, generally considered shared-memory machines, have a multiprocessor capability that allows each processor to address a number of memory banks (16 to 512) through a crossbar switch. The different processors issue separate instructions operating on data in memory that can be described as being distributed; therefore, the processors are operating in MIMD mode. Similarly, any processor in the

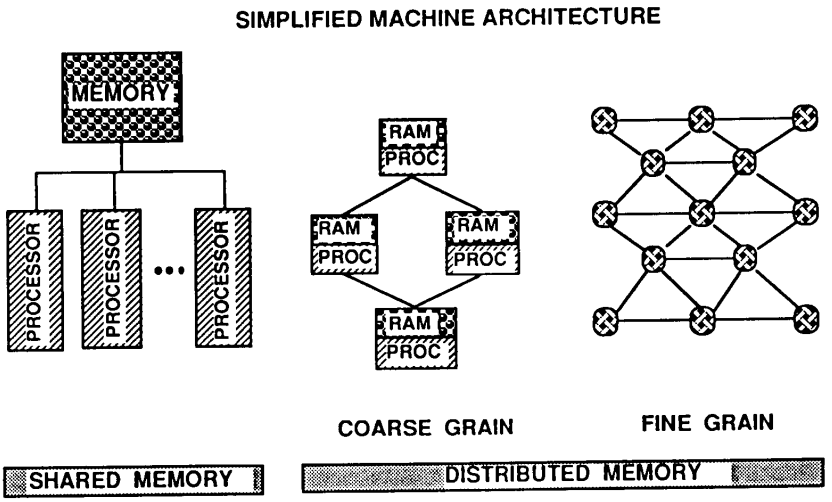

Figure 1.2 Simplified diagram of machine architecture categories

above categories may incorporate pipelining, i.e., the same instruction is operating on a multiple data stream in a "pipe," a SIMD operation. A useful overview of these concepts is given in [2, Chapter 2]. A more general description of machine organization can be found through the concepts of processor clusters and hierarchical memory [3]. In this model, clusters of distributed-memory processors can be connected together through a common bank of shared memory. Processing within clusters is performed simultaneously, with data being shared globally through the common memory banks. This organization allows memory to have a shared address space, though physically it is distributed among the processors. Currently, parallel processing is moving in the direction of machines containing thousands of processors accessing distributed memory through shared or multiple addressing. This chapter focuses on computation performed on distributed-memory machines with multiple addressing. The central issue from the user's viewpoint

is how to perform the calculations efficiently when the memory is distributed. This leads directly to the issue of problem decomposition, i.e., how to break up the problem to exploit the power of the ensemble of processors efficiently. Ideally, the programmer should not have to decide this issue; the system software, e.g., the compiler, should accomplish the task. Currently, this software does not exist, and it appears that in the near future, the user will only be assisted by high-level software in breaking up the problem at hand.

1.3 Problem Decomposition

Before a specific decomposition for each of the three scattering formulations is examined, a general programming model will be introduced [4,5]. The problem domain is decomposed into grains, with each grain corresponding to a single processor and its associated memory. Since this work examines coarse-grained machines, the memory can currently range up to 64 Mbytes at each processor, an amount of space that, for example, can store a 64-bit complex matrix of order 2000. It is essential that the grain size be fairly constant between processors. An unequal grain size, termed load imbalance, will lead to an imbalance in computation between processors, reducing the efficiency of the ensemble computation since some processors will be idle, waiting for others to complete their calculations.

The problem domain can be described as being regular or irregular, depending on the structure of the data that are mapped onto the processors. The FDTD formulation on a Cartesian grid is highly regular since the data, electric and magnetic field values, as well as the material parameters, are uniformly distributed. An integral equation solution resulting in a dense matrix is also highly regular, whereas the finite element formulation leads to an irregular problem domain due to a nonuniform mesh used to discretize the fields and geometry.

Another useful property of the three scattering formulations, as well as many other physical problems, is that they are loosely synchronous [5]. This is understood to mean that each processor is performing calculations on its own data, but must regularly stop and communicate needed data to other processors, therefore synchronizing the overall calculation from time to time. This implies that the calculations are not running in lockstep, i.e., each instruction is executing

simultaneously as in SIMD mode, or, oppositely, the calculations are not totally independent and can be performed asynchronously, never needing to synchronize. Loosely synchronous calculations fit well into a MIMD architecture. A related characteristic is that each processor executes an identical copy of the code used in the simulation, but at any given time, different processors will be executing different instructions because of different data dependencies. The processors will synchronize when data must be communicated between one or more processors to complete the next stage of calculation. For example, in an LU factorization algorithm with partial pivoting, the pivot row must be broadcast from the processor it resides in to all other processors so it can be used in the next stage of reduction. The communication requires physically moving the data between processors and produces overhead, since calculations are blocked waiting for the data to be sent and received. This overhead reduces the efficiency of the ensemble calculation.

The last characteristics to be defined are necessary in quantifying machine performance. When a problem is solved on various-sized parallel machines, fixed-grain and fixed-size problems can be defined. Fixed-grain problems refer to scaling the size of the problem geometry so that the amount of data in each processor is fixed. This scaling lies at the heart at parallel processing because the addition of more and more processors allows the solution of successively larger problems. The second scaling, fixed size, refers to the solution of a fixed-size problem on successively larger machines. The overall time needed to complete a computation will decrease as the machine size increases, but machine efficiency decreases since the ratio of communication to computation goes up as parallel portions of the algorithm are spread over more and more processors. A point of diminishing returns is reached where adding processors does not speed up the calculation.

From the above discussion, it is recognized that the goal of problem decomposition is a balance of data and computation that reduces the amount of communication needed. Load balance and minimal communication give the highest efficiency and extract the greatest performance from a parallel machine.

1.4 Finite Difference Time Domain Decomposition

The first solution to Maxwell's equations, briefly examined in this chapter, uses the FDTD method to calculate fields on a Cartesian grid. Because the electric and magnetic fields, as well as material parameters, are discretized onto a regular grid, decomposition onto a distributed-memory machine is relatively straightforward and uses the most intuitive of the three solution methods described in this chapter. The essential property of the decomposition for this method is the regularity of the spatial grid—for irregular discretizations of the FDTD solution, a decomposition method would be different and not as simple.

The FDTD algorithm [6,7] uses a leapfrog scheme in time and space to follow the evolution of electric and magnetic fields in the computational domain. This domain, consisting of the scatterer and some amount of free space, is gridded along Cartesian coordinates into unit cells and is truncated at planar boundaries into a box. Spatial derivatives of Ampere's and Faraday's laws are accomplished by 2-point centered differences on a staggered grid. Magnetic field points are staggered between the electric field points that lie on the midpoint of unit cell boundaries. At the boundaries of the computation box, an approximation of the Sommerfeld radiation condition is enforced. The same 2-point centered difference is used to perform the time derivatives on Maxwell's curl equations, therefore requiring that the electric and magnetic field points be staggered in time by one-half of a time step. Throughout the computational box, starting from time zero, the fields are advanced, updating them in time and space until a steady-state solution is found. At the boundaries of the computational box, the approximate radiation condition must also be enforced, requiring extra computation for those field points on the boundary. Similarly, the RCS is computed from the field values lying on a closed surface surrounding the scatterer. The RCS may be calculated at each time step or after some number of steps.

As outlined in Section 3, problem decomposition strives to create an even balance of data among processors while minimizing the amount of data that must be communicated. The following FDTD decomposition, which was reported in [8,9], creates this balance. Major elements of the decomposition are: 1) dividing the data among processors, 2) performing the field updates and communicating data as needed, 3) enforcing the approximate radiation condition at computa-

tional boundaries, and 4) computing the RCS on data that are divided among the processors. Decomposition of the field and geometry data is easily accomplished by successively dividing the mesh into equal-sized blocks and storing the data of each block in a separate processor. Keeping in mind the fact that field updates only need local information, i.e., one component of the field will be updated by a 2-point center difference of field values at the nearest neighbors, communication will only occur at block boundaries. It is useful to store these blocks in adjacent processors to minimize the distance data must be communicated. This storage maps naturally onto a hypercube topology. For a two-dimensional mesh topology, restricting the divisions to two of the three spatial dimensions will create blocks that map onto adjacent processors, thereby minimizing the communication path. Fig. 1.3 shows one slice of grid mapped onto four processors. Only the E_x, H_y, and H_z components of the fields are shown; arrows indicate the updating of the Ex component of the field. For a field point interior to a block, all information needed for the update is local to the processor. For a field point at a block boundary, field values in an adjacent block are needed. To expedite communicating these data, field values along a block boundary are duplicated in adjacent processors, and the entire set of values is communicated as needed. Similar figures can be drawn for the E_y and E_z components, as well as for the three components of H.

Enforcement of the approximate radiation condition requires extra computation at those field points residing on the computational box walls. These field points are obviously not load-balanced because they reside in only those processors that contain the box boundary. While these processors are performing the radiation condition enforcement, others sit idly, thereby lowering the efficiency of the overall calculation. Similarly, the RCS calculation is performed on a set of field points that surround the scatterer. These field points are not load-balanced and, as with the radiation condition enforcement, the efficiency is lowered in this part of the calculation.

Code performance for various problems is presented in [8,9]. The central result is that field updates, not including communication of the block boundary field values, comprise over 90% of the total code time for larger scaled problems. Communication in the field update is only a small fraction of the code because of the advantageous property of volume to surface scaling. Field update calculations are represented by

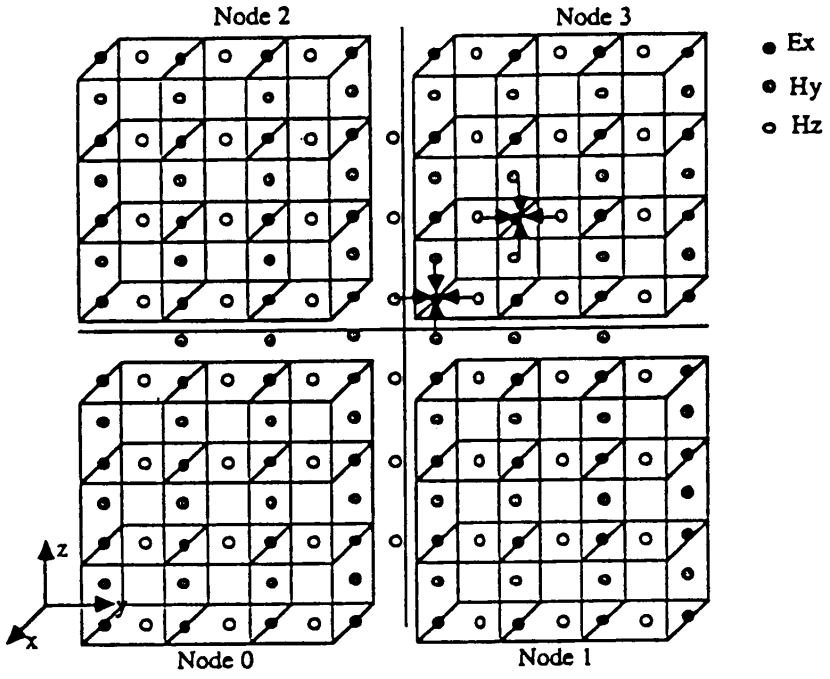

Figure 1.3 Decomposition of an FDTD grid among four processors. Arrows indicate the magnetic field components used to update the Ex component within the processors and at the boundaries of the processors (from [8, 9]).

the volume of data in a block, while communication only involves the data residing on the surface of the block. For large blocks, this ratio is large and the amount of time needed for communication is small. Similarly, radiation condition enforcement and RCS calculations are a small fraction of the total time of the code, and even though they are relatively inefficient, they do not lower the overall efficiency of the computations greatly. Overall efficiency can be 95% for scaled FDTD calculations.

1.5 Finite Element Decomposition

As a means of breaking away from using a regular grid to model arbitrarily curved structures, the finite element method using unstructured gridding is used to solve Maxwell's equations. Because of the unstructured grid, the problem domain is irregular. This domain irregularity, as well as the sparse matrix that results from discretization, leads to more demanding methods of problem decomposition.

A finite element solution begins with the generation of a solid-body model of a scatterer and the generation of a finite element mesh. As in the FDTD solution, grid nodal points are generated within penetrable regions of the scatterer and in some free-space region around the scatterer. This grid differs from the FDTD grid in that it is unstructured, conforming to the scatterer, transitioning between different densities in different parts of the computational domain (Fig. 1.4). Finite elements of some shape are meshed between nodal points, modeling the electric or magnetic fields to some degree of smoothness depending on the order of finite element used. At the boundary of the computational mesh, an approximate boundary condition is applied locally to truncate the fields.

Once the mesh is generated, a complete spatial approximation of the fields, except for the complex field amplitudes of each finite element nodal point, is specified. These complex amplitudes are found from the solution of a "weak form" of the wave equation [10]. For this discussion, the key point of this integral equation is that interactions between field values modeled by the finite element basis functions are local. The linear system consists of equations whose unknowns are the field amplitudes, and coefficients are the integrals of overlapping finite elements calculated from the weak form of the wave equation. The source, or right-hand-side vector, is known and proportional to the incident field. When written as a matrix equation, the matrix can quickly be seen to be extremely sparse since matrix entries corresponding to nonoverlapping finite elements are identically zero. The number of nonzero entries in a row will be less than 100, while the matrix rank can be over 100,000. Since the generation of matrix entries is a relatively straightforward process, the bulk of computation falls to sparse matrix solution algorithms, several types of which are available for sequential calculations [2, 11].

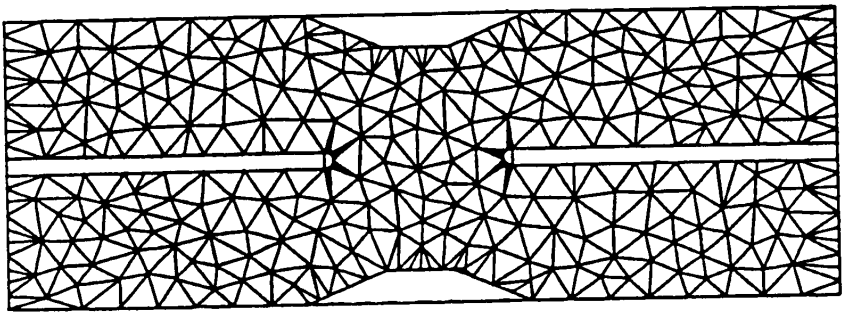

Figure 1.4 Example of a finite-element mesh illustrating an irregular problem domain.

Several steps must be considered in a parallel decomposition for the finite element solution. Data associated with the irregular mesh are divided up and stored among the processors, sparse matrix entries are calculated and also stored, the approximate boundary condition is enforced at those nodes residing on the computational mesh boundary, the sparse matrix system is solved, and finally the RCS is calculated. Because the integral associated with a matrix entry only requires local information, this calculation is completed within each processor with no communication between processors needed. Boundary condition enforcement, as well as the RCS calculation, is nearly identical to those in the FDTD calculation and suffers similar load imbalances and loss of performance. Yet, as in the FDTD solution, they are also a small fraction of the overall computation so they do not affect the overall code performance greatly [12]. The central issues in the finite element decomposition are then the mesh generation and storage onto the parallel machine, and the sparse matrix equation solution. Currently, commercially available mesh generators are typically used to model three-dimensional scatterers. They provide the graphical user interfaces necessary to model arbitrary bodies and organize the large

amounts of data generated. These data are stored and ready for use by the finite element code.

Decomposition of the large amount of mesh data among the processors must be accomplished before calculation of the sparse matrix entries begins. Because the location of nonzero entries in the matrix is directly related to the mesh—the nonzero entries in a column correspond to the overlap of one finite element with all other finite elements—it is efficient to divide the mesh spatially into a number of parts equal to the number of processors in use. In [12], an algorithm known as recursive inertial partitioning is used to successively break the mesh into parts that contain roughly equal numbers of nodal points and have minimum surface area. Equalizing the number of nodal points will cause a balance of sparse matrix entries among processors as well as a balance of the amount of computation needed to generate the entries. Minimizing the surface area will result in generally minimizing the amount of communication needed in the solution algorithm since communicated data is proportional to the surface of the blocks.

Two different types of algorithms have been used to solve the sparse system of equations. The first type, iterative algorithms, parallelizes relatively easily and scales well as the problem size increases, whereas the second type, direct algorithms, requires more effort to parallelize and has higher operation counts than the iterative algorithm. An iterative solution seeks to minimize the error in a defined functional (say, the residual) by successively updating the solution vector in a prescribed manner. It has the useful property of only working on the nonzero entries of the matrix equation, generating no new matrix elements. The algorithm revolves around matrix-vector multiplies and dot products that can be computed relatively easily in parallel. Each processor performs the needed operation on the data in its own memory, and these partial vectors or scalars are globally combined in a communication step. Performance of this type of algorithm can be found in [12].

Direct solvers decompose the original sparse matrix into lower and upper triangular sparse matrices (LU decomposition), which are then used to solve the system of equations. Several factors must be considered in a parallel direct solver. First, if an algorithm is used that requires as input only the storage of nonzero matrix entries, it will generate extra nonzero matrix entries in the computation of the upper and lower triangular decomposition (matrix fill-in). There are well-

understood sequential algorithms that minimize fill-in by reordering the matrix at each step of reduction according to some numerical criterion (e.g., see [13]). The reordering, though, would require communication of data between processors on a parallel machine, as well as continuous keeping of a balance of data between processors as the matrix entries are reshuffled—a process termed dynamic load balancing. This communication can lead to considerable overhead, reducing the efficiency of computation.

It is possible to renumber the nodal points of the original finite element grid to create a banded structure where nonzero matrix entries, as well as some number of zero matrix entries, are stored. The banded matrix system can be decomposed and stored using methods similar to the recursive inertial partitioning scheme. The LU decomposition commences, filling in a number of the zero entries as the algorithm proceeds [14]. Depending on the original grid numbering scheme, the number of zero entries that must be stored will vary, and in general, memory requirements will be greater for the banded solver.

Different parallel partitioned solvers are also being developed [11, 12, 15]. These algorithms partition the matrix into blocks and complete a solution in two steps. In the first step, factorizations are computed independently on blocks of the matrix stored in each processor. Matrix entries interior to a block are separated from boundary or "shared" entries, an operation requiring some renumbering of the nodal points. After the individual factorizations are complete, data due to the shared nodes are communicated and combined, and additional computation completes the solution. The process of communicating and combining the data may require some additional data redistribution to achieve load balance, an additional overhead.

The different solution algorithms have differing performance and memory demands on parallel machines. Iterative algorithms have the smallest storage requirements and the greatest performance, but can suffer due to the lack of convergence to accurate solutions when applied to various scattering geometries. Indeed, the convergence difficulties encountered in the application of iterative solution algorithms to integral equations, can be exacerbated in the sparse matrix solutions of the wave equation. As with dense matrix methods, useful preconditioners that are not overly costly have also not been easily found. On the other hand, the various direct methods can consistently produce accurate solutions, and one challenge in the near term is to find

efficient implementations of these algorithms.

The last point to be considered in a parallel finite element decomposition is the input of the large amount of mesh data into the parallel machine. Currently, the amount of time necessary to read these data can be quite high, at times equalling or surpassing the time needed to compute a solution. Part of this overhead is simply due to the fact that the development of input/output technology to the processors is relatively immature and advances in hardware and software will reduce the problem. A more encompassing solution is to move the step of mesh generation onto the parallel machine to both realize the machine's performance and eliminate the need to input the large amount of mesh data.

1.6 Integral Equation Decomposition

The last solution of Maxwell's equations considered in this introduction is an integral equation method. In the previous two methods, problem decomposition was accomplished by spatially dividing the meshes among processors. In the FDTD solution, geometry data and the electric and magnetic field values were divided and stored. In the finite element solution, geometry data and the sparse matrix entries that are directly related to the structure of the mesh were divided and stored. In an integral equation solution, the data to be decomposed are not the variables attached to the mesh, but rather data derived from the mesh, namely the dense impedance matrix. As will be outlined, the mesh data reside in all processors, and dividing up the dense matrix entries achieves a load balance of data and computation.

This section will examine the PATCH [16] method of moments code, a discretization of the electric field integral equation (EFIE) for conducting objects of arbitrary shape. Solid modeling of the physical object is accomplished using a mesh generator that tessellates the object's surface into triangular patches. This is the mesh that all other information is derived from. Currents on the object are modeled by pairs of the sub-domain triangular patches, a technique that results in a current representation free of line or point charges [17]. The PATCH code, as with any integral equation code, can be broken into five distinct units, which are examined separately. They consist of: 1) a geometry construction section, i.e., an algorithm that sorts through the

input mesh data and defines connectivity for the triangular patch basis functions, 2) a matrix fill routine, 3) the matrix factorization into an LU decomposition, 4) the solution of the factored matrix equation for one or more right-hand-side vectors, and 5) calculation of observable quantities such as RCS or near-field data.

The dominant computation in the integral equation solution is found in the matrix fill and matrix factorization components. Matrix fill consists of computing each of the N2 matrix entries (N is the matrix order, equal to the number of basis functions). An individual computation is, in general, a four-dimensional integration—a two-dimensional integration to find the fields from the current, and a second two-dimensional integration of this field and a testing function computing the "moments" or matrix entries. In the PATCH code, approximations are made to simplify these integrals. Matrix factorization consists of performing an LU decomposition using standard Gaussian algorithms that include partial pivoting for numerical stability.

The parallel decomposition of the integral equation solution will only examine scaled problems (Fig. 1.5). Fixed-size problems and further analysis of the scaled problem results can be found in [12]. For scaled problems, the number of unknowns (N) scales as the square of the number of processors because the dominant storage is the dense matrix containing N2 entries. This matrix will be divided among the processors.

Beginning with the geometry section of the code, it was found that the relative amount of computation needed to sort the mesh data was small, even for the largest problems attempted to date. This algorithm was not parallelized, and as seen from Fig. 1.5, the relative time needed to accomplish the geometry algorithm is quite small. In the matrix fill portion of PATCH, the EFIE is discretized into a linear set of equations as described above. Because there are N^2 matrix entries to be calculated and the problem scales as N^2, the time to fill the matrix stays constant as the problem size increases (Fig. 1.5). No communication overhead is produced because all the mesh data reside in each processor and none of the data involved in the computations need be shared between the matrix entries.

A parallel LU decomposition algorithm is used in the first stage of solving the linear system. The decomposition algorithm used is a row-based variant of Gaussian elimination with partial pivoting. In the parallel algorithm, rows of the matrix are wrapped onto the processors.

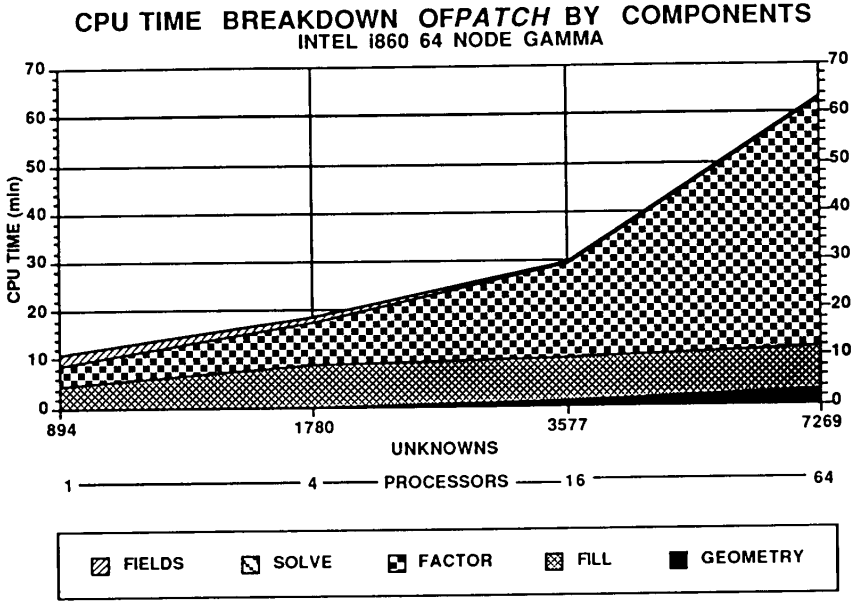

Figure 1.5 Breakdown of the PATCH code by components.

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 with all other processors. Since LU decomposition is an N3 algorithm, the execution time grows as N for scaled problems, as is seen in Fig. 1.5. Since this is the dominant part of the code at this point in its development, work continues on improving the decomposition algorithm. Use of block algorithms, as well as use of assembly-coded basic linear algebra subroutines (BLAS), is expected to greatly improve the performance [18].

To complete the solution, parallel forward reduction and backward substitution algorithms are used. Because the time needed to complete these algorithms is quite small, this part of the code (SOLVE) is lost between the FACTOR and FIELDS components in Fig. 1.5. The component labeled FIELDS in Fig. 1.5 corresponds to the calculation of electric and magnetic fields after the surface currents have been found. Since this calculation is an order N process, the relative time decreases with scaled problems of increasing size.

1.7 Software-Assisted Decomposition

From the above outline of decomposition methods for three computational techniques, it is recognized that a systematic approach to dividing the data can be readily found. Steps of decomposition involve first defining the problem-dependent data, e.g., the spatial field values or dense matrix entries, and then decomposing those data, striving to achieve a load balance of storage and computation while minimizing the amount of data communicated between processors. It has been a goal in the computer science community to automate this process by creating "parallelizing compilers" that accomplish the above steps independent of the user or class of physical problem being solved. This would be accomplished by decomposing arrays and loops in the code onto the given machine. A more realistic approach [19] begins with the premise that the compiler cannot accomplish the task independent of the user and class of physical problem. Rather, the compiler will "assist" the user in problem decomposition by supplying a small set of constructs that allow the mapping of problem data to the machine in use. The constructs are extensions to FORTRAN declaration statements that define arrays, as well as extensions to loops, that will operate on the decomposed data. Knowing the physical problem being simulated and the structure of data resulting from the algorithm in use, the user can choose the constructs needed. Low-level communication routines that are invisible to the user implement necessary data transfer between processors, hopefully in an efficient manner.

1.8 Summary

This chapter has reviewed the decomposition of three solutions to Maxwell's equations. Different decomposition properties were exhibited. The FDTD and integral equation solutions exhibit regular problem domains, while the finite element solution illustrates an irregular domain. The FDTD and finite element solutions lead to a spatial domain decomposition since the data are either the spatial field variables, as in the FDTD solution, or directly related to the spatial grid, as in the finite element solution. The integral equation solution is an example of data decomposition where the data are derived from the spatial grid. In all cases, decomposition onto a parallel machine strives to create a balance of data storage and computation among the processors, while minimizing communication overhead. For the three solution methods presented, this balance can be attained, achieving relatively high degrees of efficiency.

Acknowledgments

This overview drew on a number of sources of information. The author gratefully acknowledges interactions with the parallel processing groups at the Jet Propulsion Laboratory, and specifically the group on Research in Parallel Computational Electromagnetics, including Rob Ferraro, Jean Patterson, Paulette Liewer, Jay Parker, and Jon Partee.

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

References

[1] Flynn, M., "Some computer organizations and their effectiveness," *IEEE Trans Comput,* **C-21**, 948–960, 1972.

[2] Dongarra, J. J., I. S. Duff, D. C. Sorenson, and H. A. van der Vorst, "Solving linear systems on vector and shared memory computers," *Philadelphia, Pennsylvania, SIAM,* 1991.

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

[4] Hillis, W. D., and G. L. Steele, Jr., "Data parallel algorithms," *Comm ACM,* **29**, 1170–1183, 1986.

[5] Fox, G., M. Johnson, G. Lyzenga, S. Otto, J. Salmon, and D. Walker, *Solving Problems on Concurrent Processors,* Englewood Cliffs, New Jersey: Prentice Hall, 1988.

[6] Yee, K. S., "Numerical solution of initial boundary value problems involving Maxwell's equations in isotopic media," *IEEE Trans Antennas Propag.,* **AP-14**, 302-307, 1966.

[7] Taflove, A., K. R. Umashankar, and T. G. Jurgens, "Radar cross section of general three dimensional scatterers," *IEEE Trans Antennas Propag.,* **AP-33**, 662–666, 1985.

[8] Calalo, R. H.; T. Cwik, W. A. Imbriale, N. Jacobi, P. C. Liewer, T. G. Lockhart, G. A. Lyzenga, and J. Patterson, "Hypercube parallel architecture applied to electromagnetic scattering analysis," *IEEE Trans. Magn.,* **25**, 2898–2900, 1989.

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

[10] Silvester, P. P., and R. L. Ferrari, *Finite Elements for Electrical Engineers,* Cambridge, U.K., Cambridge University Press, 1990.

[11] Duff, I. S., A. M. Erisman, and J. K. Reid, *Direct Methods for Sparse Matrices,* Oxford, U.K., Claredon Press, 1986.

[12] Cwik, T., R. D. Ferraro, R. Hodges, N. Jacobi, P. C. Liewer, T. G. Lockhart, G. A. Lyzenga, J. W. Parker, J. Partee, J. Patterson, and D. A. Simoni, "Hypercube matrix computation task: research in parallel computational electromagnetics," *Report for 1989-1990. JPL Publication,* **91–25**, Pasadena, California: Jet Propulsion Laboratory, 1991.

[13] Osterby, O., and Z. Zlatev, "Direct methods for sparse matrices," Goos, G., and J. Hartmanis, (editors): *Lecture Notes in Computer science,* Berlin, Germany, Springer-Verlag, 1983.