3

HIGH-PERFORMANCE COMPUTING FOR FINITE ELEMENT METHODS IN LOW-FREQUENCY ELECTROMAGNETICS

G. Bedrosian

3.1 Introduction

We in the numerical analysis community have seen in the last several years an often bewildering explosion of specialized computing architectures touted by their developers as the ultimate solutions to the need for high performance for solving computationally intensive problems. Many of these architectures, such as Connection Machines and Intel iPSC Hypercubes, are parallel in one or more of the senses to be defined in Section 3.4 below, but traditional single-CPU (central processing unit) architectures still have far to go before they reach their ultimate speed limits. Within a year or two, so-called super-scalar computers with single-CPU speeds in excess of 100 MIPS (million instructions per second) and 20 Mflops (million floating point operations per

second) will become commonplace as departmental compute servers, super workstations, and even nodes in supercomputer-class parallel machines.

At the same time that improved hardware performance is attracting our attention as a method of making computationally intensive methods into routine engineering tools, we should remember that improvements in algorithms and numerical methods have historically led to greater performance gains than hardware advances alone. For example, it was commonplace in early finite element codes to use full, out-of-core matrix solvers with blocks of the matrix brought into memory from disk or tape as needed, ignoring that most of the finite element stiffness matrix is zero. Later, full solvers evolved into banded or wavefront solvers, exploiting some of the sparseness of the matrix, and finally into general sparse equation solvers that store only non-zero entries in the matrix. In each case, there was an order of magnitude increase in the speed of solution and maximum size of problem that could be solved on a given piece of computing hardware as the algorithms improved.

Unfortunately, these improved algorithms are more complex than their predecessors, both in terms of data structures and the flow of program control. We used to speak of these algorithms not vectorizing well, meaning that a significant amount of CPU time is spent performing operations not readily expressed as a sequence of repeated arithmetic operations on packed arrays of numbers in memory, e. g. linear algebra operations. We now speak of algorithms and data structures that do not map well to a particular computer architecture. If an algorithm does not map well, should we change the hardware or the algorithm? This is the key question we will address.

In numerical modeling of electromagnetic fields, the computers we use to get our solutions, fascinating though they may be in their own right, are only the means to an end: solving the problem as cost effectively as possible. We must factor into the total cost of the solution not just the cost of the initial purchase of the hardware, but hardware maintenance, software development, software support, and other operational costs as well. A new computer that theoretically delivers ten times the performance for the same cost as the current system may not be cost effective if 100 work-years of software development are required to utilize it effectively. On the other hand, it would be foolish not to take new computer architectures into account when designing new algorithms and new software, and equally foolish not to begin the

process of moving old software to new architectures if significant gains are foreseen.

This chapter will deal with some recent work directed toward more efficiently solving computationally intensive problems in low-frequency electromagnetics: electrostatic, magnetostatic, and eddy current problems. We will emphasize the finite element method for solving these problems, because in our experience this method allows the greatest flexibility and ease for the end user in setting up and running the analysis. However, we will point up some of the difficulties inherent in using finite element methods on parallel architectures. Other methods may in the end prove more powerful on parallel computers.

3.2 Finite Element Methods for Low-Frequency Problems

In this section, we summarize finite element methods for low-frequency electromagnetic problems in electrostatics, magnetostatics, and eddy currents. Most of this section will be familiar to many readers, but we have included it for the sake of completeness and to set the stage for what will follow.[1] Indeed, our presentation will strike some readers as being too detailed. However, it has been our experience that sweeping generalizations may be suitable for vendors and visionaries, but not for practitioners of numerical analysis. Mapping a numerical algorithm to a computer architecture is a matter of details; only by a thorough understanding of the details can one determine whether a numerical algorithm is suitable for a given computer architecture or, if it is not suitable, whether there is a way to modify it.

Because much of the motivation behind high-performance computing for low-frequency problems is to make three-dimensional analysis a routine design tool for engineers — as two-dimensional analysis is now — we will emphasize methods that are applicable for three-dimensional analysis, where by "three-dimensional" we mean both the electromagnetic fields themselves and the models we wish to analyze. We begin with Maxwell's equations for macroscopic fields in rationalized m. k. s.

[1] Introductions to the finite element method in general and to finite elements as applied to low-frequency electric and magnetic fields can be found in [6] and [23] respectively, as well as in many other standard references.

units, and their subsequent mathematical development for the special cases of electrostatic, magnetostatic, and eddy current problems. Following that, we will present a brief overview of the finite element method. In subsequent sections, we will discuss the mapping of this method to several high-performance computer architectures.

The derivation of finite element methods for low-frequency electromagnetic problems begins with Maxwell's equations in their nonrelativistic differential form for macroscopic media [1]:

$$\nabla \times H = J + \frac{\partial D}{\partial t} \tag{1}$$

$$\nabla \cdot B = 0 \tag{2}$$

$$\nabla \times E = -\frac{\partial B}{\partial t} \tag{3}$$

$$\nabla \cdot D = \rho \tag{4}$$

where (in order of appearance above) H is the magnetic field, J is the transport current density, D is the displacement field, t is time, B is the magnetic induction field or flux density, E is the electric field, and ρ is the free charge density.

In static problems, the time derivative terms in (1) and (3) drop out, decoupling the electric and magnetic fields. If E is given as the negative gradient of an electrostatic potential, Φ ,

$$E = -\nabla \Phi \tag{5}$$

and the displacement field, D, is related to E through the material permittivity, ε,

$$D = \varepsilon E \tag{6}$$

then (3) is automatically satisfied and (4) becomes

$$\nabla \cdot (\varepsilon \nabla \Phi) = -\rho$$

Because the curl of H is not identically zero in (1), H cannot be derived in general from the gradient of a scalar potential alone, even in static problems. One method to account for this is to solve instead for a magnetic vector potential, A, where B is the curl of A :

$$B = \nabla \times A \tag{8}$$

H is related to B through the material reluctivity, ν, where ν is the inverse of μ, the magnetic permeability [2]

$$H = \nu B \tag{9}$$

Given (8), (2) is satisfied identically. Combining (8) and (9), (1) becomes

$$\nabla \times (\nu \nabla \times A) = J \tag{10}$$

One serious difficulty with (8) is the problem of gauge. Briefly, the flux density in (8) is unaffected by adding the gradient of any arbitrary scalar function to A. This leads to instabilities and/or singularities in the solution of (10). The traditional resolution of this problem has been to introduce a gauge to enforce uniqueness of A; the so-called Coulomb gauge is usual in low-frequency problems:

$$\nabla \cdot A = 0 \tag{11}$$

Other gauges are possible as well [2]. However, enforcing a gauge in a numerical algorithm without disturbing the accuracy of the solution of (10) is difficult at best.

Problems with uniqueness aside, solving for three components of the unknown vector, A, in a three-dimensional finite element problem is expensive numerically. One way of avoiding the problems inherent in solving for a vector unknown is to cast the magnetostatic problem in terms of a reduced magnetic scalar potential, Φ_m :

$$H = H_s - \nabla \Phi_m \tag{12}$$

H_s is the magnetic field solution in free space from a given source current distribution, J_s :

$$\nabla \times H_s = J_s \tag{13}$$

$$\nabla \cdot H_s = 0 \tag{14}$$

H_s can be calculated by any one of several classical methods for particular cases, or by the law of Biot-Savart for the general case [1]:

$$H_s(r) = \frac{1}{4\pi} \int \int \int \frac{J_s(r') \times (r - r')}{|r - r'|^3} d^3V' \tag{15}$$

[2] In practical magnetostatic problems, the relationship between H and B is highly nonlinear and exhibits hysteretic effects as well. We will not deal with these complications here.

Since we are now solving for H, we express B in terms of the magnetic field through the material permeability, μ :

$$B = \mu H \tag{16}$$

Maxwell's magnetic curl equation (1) is identically satisfied by (12) and (13). Combining (12), (14), and (16), Maxwell's equation for zero divergence of magnetic flux density (2) becomes

$$\nabla \cdot (\mu \nabla \Phi_m) = \nabla \cdot (\mu H_s) = (\nabla \mu) \cdot H_s \tag{17}$$

Note that (17) is mathematically identical to (7), with the right-hand side serving as a fictitious magnetic charge density: a volume charge density where μ varies continuously within a nonlinear material, and a surface charge density at material interfaces where μ is discontinuous.

We have eliminated problems with gauge and reduced three degrees of freedom per node to one by introducing a reduced magnetic scalar potential, but nothing comes free. The reduced scalar potential solution suffers from cancellation effects in regions with high μ, where the two right-hand terms in (12) have nearly equal magnitudes but opposite directions. For many problems of interest, however, the effect of the cancellation errors inside high-μ regions on the fields outside high-μ regions is tolerable, provided the surface charge terms in (17) are correctly and accurately handled by the numerical analysis. Another method of handling the cancellation is to solve for a full magnetic scalar potential, which is possible only if all source currents are represented as current sheets and branch cuts (discontinuities) in magnetic scalar potential are introduced at the sheet surfaces.

In eddy current problems, the displacement current term in (1) is neglected and (4) is usually ignored in favor of enforcing zero divergence of transport current. (More about this later.) Currents in conductors are given by Ohm's law, so that (1) becomes

$$\nabla \times H = \sigma E \tag{18}$$

where σ is the material conductivity.

Like magnetostatic problems, eddy current problems can be solved with a magnetic vector potential or a reduced scalar potential. We will consider a vector potential formulation first. With (8) as before, it is evident that

$$E = -\frac{\partial A}{\partial t} - \nabla \Phi \tag{19}$$

satisfies (3) for any choice of Φ. Combining (8), (9), and (19), (18) becomes

$$\nabla \times (\nu \nabla \times A) + \sigma \frac{\partial A}{\partial t} = -\sigma \nabla \Phi = J_s \tag{20}$$

It should be noted in (20) that the source current term, J_s, is included as a convenience for analysis in regions where the current is known and eddy currents do not exist, such as stranded and transposed wire. Such regions are modeled as source currents with zero conductivity.

Non-divergence of the transport current must be enforced in order for the solutions of (20) to be physically meaningful. In particular, currents should not exit the interface surfaces between conductors and non-conductors. Taking the divergence of (20) and setting it to zero, we obtain

$$\nabla \cdot (\nu \nabla \Phi) = -\nabla \cdot \left(\sigma \frac{\partial A}{\partial t} \right) = -\frac{\partial}{\partial t} \nabla \cdot (\sigma A) \tag{21}$$

Note that (21) can be solved only in materials with non-zero conductivity; Φ is indeterminate in non-conducting regions.

It still remains to enforce a gauge. A common gauge choice (often made implicitly) is to set $\Phi = 0$, i.e. drop the scalar potential completely. This choice appears to have the advantage of removing one unknown scalar at each node in conductors, but it has a serious problem. Although the finite element formulation will be discussed in more detail later in the chapter, we must invoke here some finite element properties in order to understand the problem with this gauge.

When $\Phi = 0$, E and A are identical except for time differentiation and a sign change, neither of which affect continuity properties. This leads to difficulty because the normal component of E should be discontinuous across material interfaces, but the standard nodal finite element basis forces continuity of the normal component of A and hence E as well. Edge elements have been proposed as a solution to the dilemma of discontinuous normal E [3,4]. In this approach, one solves for the tangential component of A along each edge in a tetrahedral finite element mesh, rather than three components of A at each node. This results in twice as many degrees of freedom as compared to the node-based formulation for the same finite element mesh.[3]

[3] There are fewer non-zero entries per row of the matrix as compared with node-based finite elements. If an iterative solver is used which stores only non-zero matrix elements, this may somewhat (but not fully) compensate for the larger number of unknowns.

Enforcing continuity of all components of A is incorrect only at material interfaces. Within material regions, exact continuity of all components of A is desirable. Furthermore, it is desirable to enforce continuity of the normal component of (σA) at material interfaces to ensure current continuity. These conditions can be enforced by tearing the finite element mesh along material boundaries and enforcing the correct continuity conditions by auxiliary equations [5].

Even with either of the above solutions to the problem of the discontinuity of the normal component of E, the difficulties with gauge are not over. Dropping Φ from (21), the gauge is

$$\nabla \cdot (\sigma A) = 0 \tag{22}$$

This gauge works in conductors; in fact, (22) follows naturally from the solution of (20) when $\Phi = 0$. But (22) vanishes in non-conductors. Since most eddy current problems involve combinations of conducting and non-conducting regions, it is necessary to introduce a second gauge, such as the Coulomb gauge (11), in non-conducting regions. If edge elements are used, the continuity of the normal component of current density is not enforced locally. If the standard nodal finite element basis is used, enforcing the Coulomb gauge through a penalty function or by other means is numerically difficult. These problems lead us to consider an alternative formulation using a reduced scalar potential.

Let H be written as the sum of three terms:

$$H = H_s + H_e - \nabla \Phi_m \tag{23}$$

H_s and Φ_m are defined as before for magnetostatic problems. H_e is the part of the field whose curl gives the eddy current density:

$$\nabla \times H_e = J_e = \sigma E \tag{24}$$

H_e is defined to be zero outside of conducting regions. Because of this, it is necessary to ensure that the conducting region(s) in the problem are simply connected; i.e. have no holes. Otherwise, Ampere's law is violated for a conductor that has induced currents circulating around the hole. The simplest example of an object with this problem is a torus. There are several ways to deal with conductors that have holes, the simplest being to fill the holes with a fictitious material whose conductivity is much smaller than the conductivity of the original conductor.

Combining (23), (24), and (3), we obtain the curl-curl equation for H_e :

$$\nabla \times \left(\frac{1}{\sigma}\nabla \times H_e\right) + \mu\frac{\partial H_e}{\partial t} = -\mu\left(\frac{\partial H_s}{\partial t} - \frac{\partial \nabla \Phi_m}{\partial t}\right) \tag{25}$$

(Note that (25) only applies inside conductors.) Ignoring for the moment that Φ_m is unknown, (25) is mathematically identical to (20).

The complementary div-grad equation for Φ_m is obtained by combining (23) and (14) with (2):

$$\nabla \cdot (\mu \nabla \Phi_m) = (\nabla \mu) \cdot H_s + (\nabla \mu) \cdot H_e + \mu \nabla \cdot H_e \tag{26}$$

Ignoring for the moment that H_e is unknown, (26) is mathematically equivalent to (17). Continuity of the normal component of eddy current density is guaranteed if the tangential components of H_e are continuous; these components are set to zero as boundary conditions at interfaces between conductors and non-conductors so that eddy currents cannot enter or exit through these interfaces. However, the normal component of H_e jumps discontinuously to zero at interfaces between conductors and non-conductors (because H_e is defined to be zero in non-conductors), giving rise to surface charge contributions on the right-hand side of (26).

Equations (25) and (26) can be solved together, or they can be solved iteratively using adaptive relaxation. The iterative method is especially useful when H_e is a perturbation of the total field, as is the case in the interaction of low-frequency magnetic fields with poor conductors, i.e. conductors whose skin depths at the frequencies of interest are large compared with their geometric sizes.

From the preceding discussions, it is clear that we need to solve two basic types of partial differential equations in low-frequency problems: the div-grad equation (7) and the curl-curl equation (20). We will now turn our attention to solving these equations numerically.

The fundamental choice we make before beginning a numerical solution is the mathematical representation of the solution, for this will usually dictate — or at least limit — the type of numerical analysis method we can use. If, for example, we are interested in the solution only at a set of regularly spaced points, then a finite difference scheme that converts (7) and (20) to difference operators would be appropriate as a solution method. However, we are interested in the numerical solution everywhere in space, not just at a set of discrete points. Furthermore, the objects we wish to model can be geometrically complex

and the solution basis set may need to be more accurate in some regions than in others. For these reasons, we seek a mathematical representation of the solution that conforms to geometrically complex objects, allows local improvement of numerical accuracy, and interpolates the solution over the entire domain. It should also be capable of handling nonlinear material properties that are essential in low-frequency magnetics.

A numerical representation scheme that satisfies all of the above criteria is the finite element method [6]. In this method, the solution space is divided into small sub-volumes called finite elements, each element being a simple geometric shape such as a tetrahedron, triangular prism, or hexahedron (brick). Within each element, any continuous function, f, is interpolated as the weighted sum of finite element shape or basis functions, N_i :

$$f(x,y,z) = \sum_{i=1}^{\text{nodes}} f_i N_i(x,y,z) \tag{27}$$

Each shape function is associated with a point in space called a node, such that the shape function has a value of one at its associated node and zero at all other nodes:

$$N_i(x_j,y_j,z_j) = \left\{ \begin{array}{ll} 1, & i=j \\ 0, & i \neq j \end{array} \right\} \tag{28}$$

Finite element shape functions also have the property of being identically zero in any element not containing their respective nodes. This property is often called local support.

Vertices of finite elements are nodal points. In the simplest three-dimensional element, the first-order tetrahedron, there are four shape functions that are linear (order one) functions of the geometric coordinates (x,y,z) satisfying (28) for each of the four vertices. Elements with higher-order shape functions (order greater than one) have nodes at additional locations. In any case, the shape functions always sum to one within the element, including its boundaries:

$$1 = \sum_{i=1}^{\text{nodes}} N_i(x,y,z) \tag{29}$$

Elements can have different sizes and aspect ratios depending on the locations of their vertices. Elements with higher-order shape functions can also have curved edges, a property that is helpful in matching

the finite element mesh accurately to the underlying geometry of the problem if curved geometric surfaces are involved. In order to handle these cases with mathematical generality, it is best to introduce a set of local (dimensionless) coordinates for each element that are mapped to the global (physical) coordinates of the problem. The local coordinates of an element's nodes are independent of the actual size and orientation of the element. We will refer to these local coordinates as u_1, u_2, and u_3.

The finite element shape functions can be written as functions of local coordinates. (28) and (29) still hold, but are now expressed in terms of the local coordinates:

$$N_i(u_{1j}, u_{2j}, u_{3j}) = \left\{ \begin{array}{ll} 1, & i = j \\ 0, & i \neq j \end{array} \right\} \tag{30}$$

$$1 = \sum_{i=1}^{\text{nodes}} N_i(u_1,\ u_2,\ u_3) \tag{31}$$

The shape functions can be written as polynomials in the local coordinates; these polynomials are invariant with respect to the element's actual size and location in the global coordinate system. For example, Fig. 3.1 shows a first-order hexahedral element.

The global coordinates within a distorted element can be interpolated using the same shape functions. Elements of this kind are called isoparametric [6]. The global coordinates can be found from the local coordinates at any point in the element by weighting the shape functions with the global coordinate values at the respective nodes:

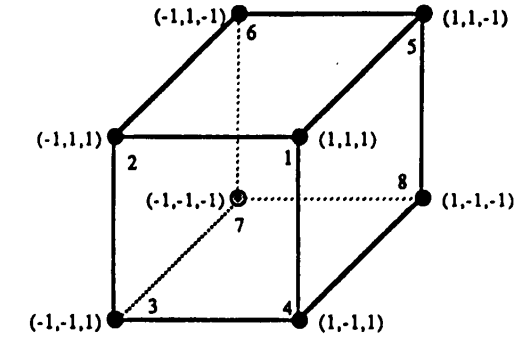
$$x = \sum_{i=1}^{\text{nodes}} x_i N_i(u_1,\ u_2,\ u_3) \tag{32}$$

$$y = \sum_{i=1}^{\text{nodes}} y_i N_i(u_1,\ u_2,\ u_3) \tag{33}$$

$$z = \sum_{i=1}^{\text{nodes}} z_i N_i(u_1,\ u_2,\ u_3) \tag{34}$$

The partial derivatives of the global coordinates with respect to the local coordinates are found by straightforward differentiation:

$$\frac{\partial x}{\partial u_k} = \sum_{i=1}^{\text{nodes}} x_i \frac{\partial N_i}{\partial u_k} \tag{35}$$



$$N_1 = \frac{(1+u_1)(1+u_2)(1+u_3)}{8} \qquad N_5 = \frac{(1+u_1)(1+u_2)(1-u_3)}{8}$$

$$N_2 = \frac{(1-u_1)(1+u_2)(1+u_3)}{8} \qquad N_6 = \frac{(1-u_1)(1+u_2)(1-u_3)}{8}$$

$$N_3 = \frac{(1-u_1)(1-u_2)(1+u_3)}{8} \qquad N_7 = \frac{(1-u_1)(1-u_2)(1-u_3)}{8}$$

$$N_4 = \frac{(1+u_1)(1-u_2)(1+u_3)}{8} \qquad N_8 = \frac{(1+u_1)(1-u_2)(1-u_3)}{8}$$

Figure 3.1 Shape functions for a first-order hexhedron.

$$\frac{\partial y}{\partial u_k} = \sum_{i=1}^{\text{nodes}} y_i \frac{\partial N_i}{\partial u_k} \tag{36}$$

$$\frac{\partial z}{\partial u_k} = \sum_{i=1}^{\text{nodes}} z_i \frac{\partial N_i}{\partial u_k} \tag{37}$$

The Jacobian matrix for the transformation from local coordinates in an element to global coordinates is given by

$$[J] = \begin{bmatrix} \frac{\partial x}{\partial u_1} & \frac{\partial x}{\partial u_2} & \frac{\partial x}{\partial u_3} \\ \frac{\partial y}{\partial u_1} & \frac{\partial y}{\partial u_2} & \frac{\partial y}{\partial u_3} \\ \frac{\partial z}{\partial u_1} & \frac{\partial z}{\partial u_2} & \frac{\partial z}{\partial u_3} \end{bmatrix} \tag{38}$$

In first-order tetrahedral elements, the Jacobian matrix is constant. In other three-dimensional elements, the Jacobian matrix can vary from point to point depending on the placement of the nodes.

Partial derivatives of the shape functions with respect to local coordinates are independent of the locations of the nodes in global

space. They can be pre-computed and stored at a given fixed set of local coordinate points within the element, for example Gaussian integration points. However, we will see below that the finite element matrix equation depends on terms involving partial derivatives of the shape functions with respect to global coordinates. To compute these derivatives, one first inverts the 3×3 Jacobian matrix. The global derivatives are then found by matrix multiplication of the transposed local derivative vector times the inverse Jacobian matrix:

$$\left[\frac{\partial N_i}{\partial x}\frac{\partial N_i}{\partial y}\frac{\partial N_i}{\partial z}\right] = \left[\frac{\partial N_i}{\partial u_1}\frac{\partial N_i}{\partial u_2}\frac{\partial N_i}{\partial u_3}\right][J]^{-1} = \left[\frac{\partial N_i}{\partial u_1}\frac{\partial N_i}{\partial u_2}\frac{\partial N_i}{\partial u_3}\right]\begin{bmatrix} \frac{\partial u_1}{\partial x} & \frac{\partial u_1}{\partial y} & \frac{\partial u_1}{\partial z} \\ \frac{\partial u_2}{\partial x} & \frac{\partial u_2}{\partial y} & \frac{\partial u_2}{\partial z} \\ \frac{\partial u_3}{\partial x} & \frac{\partial u_3}{\partial y} & \frac{\partial u_3}{\partial z} \end{bmatrix} \quad (39)$$

We will see later how floating point hardware originally designed for graphics transformations can be applied to speed up this computation.

We will now use the computational machinery of isoparametric finite elements to solve Poisson's equation (7) numerically. The solution is expressed as a linear combination of the finite element basis (shape) functions:

$$\Phi = \sum \Phi_i N_i \quad (40)$$

Some of the coefficients Φ_i in (40) are known *a priori* by virtue of being Dirichlet boundary values, that are of course required for (7) to have a unique solution. The other coefficients are degrees of freedom.

Consider a vector space, V, whose basis set is the set of all finite element shape functions, N_j, associated with nodes that are not Dirichlet boundary nodes. The inner product of f and g, two arbitrary functions in this space, can be defined as an integral over the domain, V :

$$\langle f|g\rangle = \int\int\int fg d^3V \quad (41)$$

We can rewrite (40) in terms of basis vectors in V as

$$\Phi = \sum \Phi_j N_j + \Phi_b \quad (42)$$

where Φ_b accounts for the known potentials on the Dirichlet boundary, whose associated shape functions are outside of V. Given this definition, we can also construct an error functional: a measure of how much

(42) deviates from the exact solution of (7) at any given point:

$$r(\Phi) = \nabla \cdot (\epsilon \nabla \Phi) + \rho = \sum \Phi_j \nabla \cdot (\epsilon \nabla N_j) + \nabla \cdot (\epsilon \nabla \Phi_b) + \rho \tag{43}$$

The best possible numerical solution in V minimizes $\langle r|r \rangle$, which is mathematically equivalent to making r orthogonal to V :

$$0 = \langle N_k | r(\Phi) \rangle = \int \int \int N_k \left[\sum \Phi_j \nabla \cdot (\epsilon \nabla N_j) + \nabla \cdot (\epsilon \nabla \Phi_b) + \rho \right] d^3V \tag{44}$$

Using a vector identity and the Divergence Theorem, (44) can be reformulated as

$$\int \int \int \nabla N_k \cdot \left[\sum \Phi_j (\epsilon \nabla N_j) + (\epsilon \nabla \Phi_b) \right] d^3V = \int \int \int N_k \rho d^3V + \int \int N_k \sum \Phi_j (\epsilon \nabla N_j) \cdot d^2S + \int \int N_k (\epsilon \nabla \Phi_b) \cdot d^2S \tag{45}$$

where the surface integrals in (45) are at the boundaries of the domain.[4] The second surface integral in (45) is always zero because the shape function, N_k, is identically zero on any boundary where the solution is known, i.e. any Dirichlet boundary, by the definition of V and the properties of finite element basis functions. The first surface integral in (45) is often neglected as well. This is tantamount to setting the outgoing or incoming flux at the boundary, F_b, to zero:

$$\sum \Phi_j (\epsilon \nabla N_j) = \epsilon \nabla \Phi = F_b = 0 \tag{46}$$

(46) is a homogeneous Neumann boundary condition, which is the natural boundary condition of a scalar finite element formulation: If one does nothing to set a boundary condition, one gets (46) by default. If one instead wishes to set the outgoing flux to a known value, i.e. $F_b \neq 0$, (45) becomes

$$\sum \Phi_j \int \int \int \nabla N_k \cdot (\epsilon \nabla N_j) d^3V = \int \int \int N_k \rho d^3V + \int \int N_k F_b \cdot d^2S - \int \int \int \nabla N_k \cdot (\epsilon \nabla \Phi_b) d^3V \tag{47}$$

[4] In electromagnetic problems, the outer boundary often extends to infinity, a complication beyond the scope of this chapter.

where we have moved all source terms to the right-hand side. Note that Neumann boundary conditions are satisfied only in a least residual sense, unlike Dirichlet boundary conditions that are satisfied exactly by fixing the nodal potentials on the boundary.

Equation (47) is a symmetric, positive-definite matrix equation. Because of the local support property of finite elements, it is also sparse. In fact, it is easy to show that the average number of non-zero matrix elements per row in (47) depends only on the order of finite elements used and not on the total number of elements in the entire grid. The total size of the matrix, counting only non-zero matrix elements, scales linearly with the number of degrees of freedom. This property is one of the chief advantages of the finite element method when compared with the boundary element method, for which the matrix size scales as the square of the number of degrees of freedom. Unfortunately, methods for calculating, storing, and solving sparse matrix equations are more complex than their full matrix counterparts.

Similar finite element methods can be applied to the curl-curl vector equation (20). Let the unknown vector potential, A, be expressed as a weighted sum of vector basis functions, N_j, plus a boundary term, A_b :

$$A = \Sigma A_j N_j + A_b \tag{48}$$

This expression is similar to its scalar counterpart, (42). The vector shape functions can be either edge based or node based, as previously discussed. Using mathematical arguments similar to the scalar case, we arrive at the following equation to minimize the error residual:

$$\int\int\int N_k \cdot \nabla \times \nu(\sum A_j \nabla \times N_j + A_b) d^3V + \int\int\int N_k \cdot \sigma \left(\sum \frac{\partial A_j}{\partial t} N_j + \frac{\partial A_b}{\partial t}\right) d^3V = \int\int\int N_k \cdot J_s d^3V \tag{49}$$

Note that time derivatives operate on the scalar weights, A_j, in (48), while the curls operate on the vector basis functions. If the problem is solved in the frequency domain, the time derivatives translate to multiplication by ($j\omega$) (for $e^{j\omega t}$ time dependence), while in the time domain it is usual to apply finite differences to model the time derivatives while retaining finite elements for the space derivatives.

After manipulation using vector identities and the Divergence Theorem, and taking all sources to the right-hand side, we obtain

$$\sum A_j \int\int\int \nabla \times N_k \cdot \nu \nabla \times N_j d^3V + \sum \frac{\partial A_j}{\partial t} \int\int\int N_k \cdot \sigma N_j d^3V - \sum A_j \int\int N_k \times \nu \nabla \times N_j \cdot d^2S = \int\int N_k \times \nu \nabla \times A_b \cdot d^2S + \int\int\int N_k \cdot \left(J_s - \sigma \frac{\partial A_b}{\partial t} \right) d^3V \tag{50}$$

Unlike (47), it is not always possible to eliminate the surface integrals in (50). However, (50) is in other respects similar to (47) in its computational requirements, albeit more complicated. Both (47) and (50) require numerical integration of derivatives of shape functions and the solution of sparse matrix equations. And of course, both require a finite element mesh conforming to the geometry of the problem, with local mesh refinement as needed for accuracy. These are the computational paradigms of finite element analysis, which we will discuss at length in the next section.

3.3 Finite Element Paradigms

A paradigm is "an oustandingly clear or typical example or archetype" [7]. When used in relation to computer architectures, a paradigm is an archetypical calculation for which an architecture may or may not be well suited. In the traditional methodology of finite element analysis, implicitly influenced by the ultimate implementation of the algorithms on single-CPU machines, the problem proceeds in an orderly and modular fashion:

1. Generate and save the finite element mesh. ("Pre-Processing")
2. Calculate and store the coefficients of the matrix and the right-hand side(s) of (47) or (50).
3. Solve the matrix equation for one or more right-hand sides and save the solution(s) to a file.
4. Calculate and/or display quantities of interest that can be derived from the solution(s). ("Post-Processing")

In this section, we will examine each of these steps to determine their computational requirements. Although our point of departure

will be to take each step independently in a single-CPU approach, it is worthwhile to consider the entire process as a whole: a restructuring of the problem to provide a better fit to one or more parallel architectures. The goal, of course, is ultimately an accurate and efficient numerical solution to an electromagnetic problem; the particular numerical methods are means to that end. Some methods might be less efficient on single-CPU machines, but well suited for multiple-CPU machines. Therefore, we will examine an alternative formulation as well as the finite element steps as usually implemented on single-CPU computers.

Pre-processing consists of defining the geometry of the region of interest; specifying the sources, boundary conditions, and material properties; and finally subdividing the geometry into finite elements. Of these, subdividing the geometry or *mesh generation* stands to gain the most from high-performance computing because it is the most computationally expensive if done automatically.

There are three basic types of mesh generation schemes. The first type, manual mesh generation, calls upon the user to input manually the nodal coordinates and element connectivity (node number) lists. This method is rarely used today, for obvious reasons.

The second type has often been imprecisely termed *automatic* mesh generation because it places nodes and generates elements without individual user inputs by constructing a finite element mesh in a fixed topological form (usually rectangular) and warping it to fit a geometric outline. However, we will use the more accurate term, *mapped* mesh generation, in this chapter. The advantages of mapped mesh generation are its speed and the ability of the subsequent analysis to capitalize on the rectangular numbering scheme of the resulting mesh. The disadvantage of mapped mesh generation is the near impossibility of writing a general algorithm to subdivide a geometry of arbitrary complexity into mappable subregions. Fig. 3.2 shows a topologically rectangular two-dimensional grid that has been mapped to conform to an L-shaped region with one curved side.

The third type of mesh generator has been called *fully automatic,* but here we will simply use the term *automatic,* meaning a mesh generator that is capable of taking as input a geometric description of arbitrary complexity and generating a finite element mesh with no user input, with the exception of optional guidelines for mesh density to override the automatic defaults. An important feature of the resulting

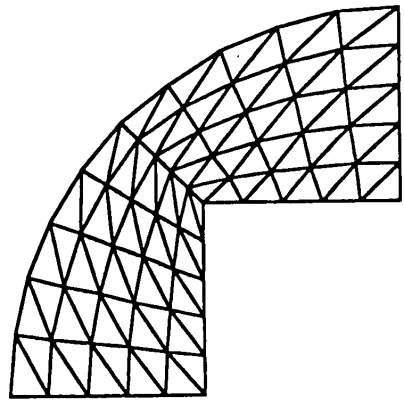


Figure 3.2 Mapped Mesh.

mesh is that it contains elements of varying sizes as required to track the geometric features and/or provide accurate solutions in areas of interest. Several schemes for two-dimensional automatic mesh generation are currently in use, and work is progressing on three-dimensional schemes [8]. They share the following characteristics:

1. The resulting mesh does not conform to a simple topological structure.
2. Element types include triangles (2D) and tetrahedra (3D) of necessity, particularly in locations requiring mesh density transitions and/or tracking of geometric boundaries.
3. In the absence of explicit renumbering, node and element numbers are arbitrary.
4. The mesh generation algorithms use additional data structures beyond simple nodal coordinates and element node lists.
5. The algorithms spend a significant amount of CPU time doing geometric calculations: point inside/outside a geometric region, distance of a point to a geometric edge, intersections, etc.
6. Many (not all) algorithms rely on recursive splitting techniques.

Fig. 3.3 shows the same L-shaped geometry meshed with an automatic recursive subdivision algorithm.

Using automatic mesh generation schemes can be as computationally expensive as solving the finite element problem itself, if not more

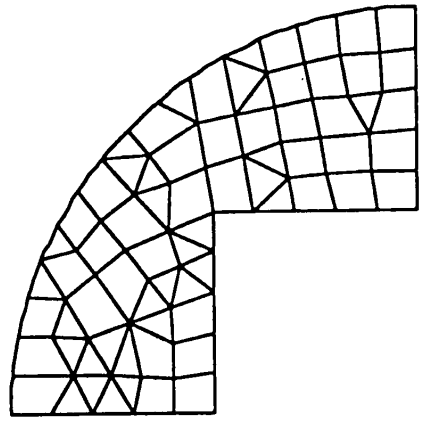


Figure 3.3 Automatic mesh.

so. For example, we have used the method of [8] to create a three-dimensional mesh with approximately 64,000 nodes in about eight hours of CPU time on a Digital Equipment minicomputer. A scalar potential finite element analysis on a mesh of the same size on the same computer takes only about four hours. We will see in the next section what would be required to run the mesh generation calculations in parallel.

The process of performing numerical integrations to determine the matrix coefficients of the finite element variables in (47) or (50) is known as matrix assembly. The matrix for the entire problem, sometimes called the global stiffness matrix, $[S]$, is the sum of contributions from each element, sometimes called element stiffness matrices. Element stiffness matrices are usually calculated in a condensed form, using local element vertex numbers. The element's contribution is then scattered into the global matrix rows and columns according to the global node numbers corresponding to the element's vertices. [5] Fig. 3.4 is a schematic diagram of the process of assembling the global stiffness matrix.

The element stiffness values, E_{kj}, in Fig. 3.4, are computed by

[5] In the discussion to follow, we will ignore for simplicity the effect of boundary conditions and we will assume that there is only one unknown quantity per node.

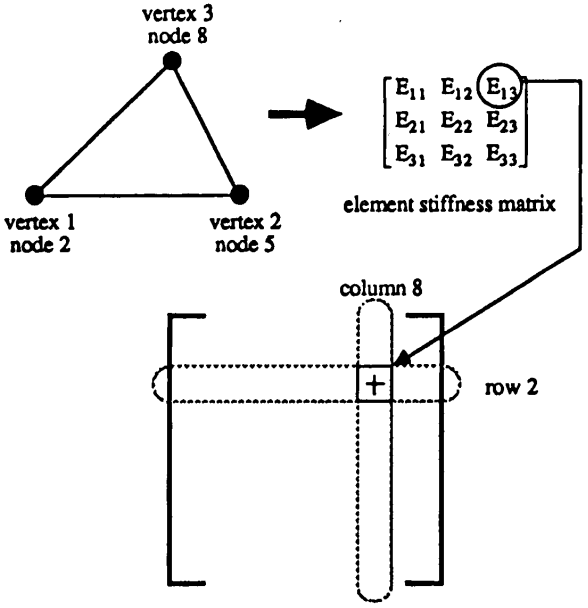


Figure 3.4 Assembly of global stiffness matrix.

integrating the term on the left-hand side of (47) over one element:

$$E_{kj} = \int\int\int_{\text{element}} \nabla N_k \cdot \epsilon \nabla N_j d^3V \tag{51}$$

The integral over the entire volume is the sum of the contributions from all elements.

The global stiffness matrix, $[S]$, is a sparse matrix because $S_{ij} = 0$ unless nodes i and j have at least one element in common. As mentioned in the previous section, the average number of non-zero matrix coefficients per row of $[S]$ depends only on the types of elements used, not on the total number of nodes in the problem. Because $[S]$ is sparse, it is wasteful to store the entire matrix as a simple FORTRAN double-dimensioned array. One alternative is to store $[S]$ as a banded matrix after renumbering the nodes to reduce bandwidth. Even then, a considerable amount of storage is wasted because of the high fraction of trapped zeros within the band, especially for three-dimensional problems. Another approach is to use a data structure that stores only

the non-zero coefficients of $[S]$. This saves storage, but requires an iterative method for solution ($[S]$ fills in otherwise).

Another complication arises in electromagnetics problems when matching an exterior solution that goes to infinity at the outer boundary of the mesh. Several techniques can be used, but their details are beyond the scope of this chapter. For some of them, the effect is the same as adding one large element matrix with terms for each pair of nodes on the boundary. If the nodes on the boundary are numbered in a cluster at the beginning of the node list, then $[S]$ becomes a hybrid sparse/dense matrix, with a fully dense upper left-hand corner. Finding efficient solution methods for such matrices is a current research topic. Another method is the absorbing boundary condition, which replaces an exact expression for the exterior field with a local approximation that preserves the sparseness of $[S]$. Choice of the best solution technique will determine the data structure used to store $[S]$, that will in turn affect parallel decomposition of the assembly step, as we will see later.

The matrix solution method that is best for a finite element run depends on several factors, the most important of which are

1. Ability to exploit symmetry, sparsity, and/or bandwidth to reduce the amount of memory needed to store the matrix and ancillary data.
2. Scaling of solution time with the number of degrees of freedom.
3. Tolerance of ill-conditioning.
4. Need for repeated solutions and/or obtaining an explicit inverse.
5. Suitability for exploiting available parallel computing hardware.

There are two general classes of methods: direct solvers and iterative solvers. Direct solvers include all methods that either generate an explicit inverse, or factor the matrix into the product of two or three matrices that are easier to solve than the original. If the original matrix equation is

$$[S]x = y \tag{52}$$

then the decomposed matrix equation

$$[S]x = [L][U]x = y \tag{53}$$

where $[L]$ is lower triangular and $[U]$ is upper triangular, can be solved by forward elimination

$$[L]z = y \tag{54}$$

followed by back substitution

$$[U]x = z \tag{55}$$

If $[S]^{-1}$ is required, it can be obtained by inverting $[L]$ and $[U]$:

$$[S]^{-1} = [U]^{-1}[L]^{-1} \tag{56}$$

When $[S]$ is symmetric, $[U]$ is the transpose of $[L]$ and the decomposition process is called Cholesky decomposition.

The advantages of full and banded direct solvers lie in their simplicity. The matrix coefficients can be stored in FORTRAN arrays with straightforward indexing, suitable for conventional vector processing. If repeated solutions are required, the incremental cost is low because the difficult part (decomposition) need be done only once. Finally, for full matrix solvers there is no memory penalty in obtaining an explicit inverse because the complete inversion process can be done step-by-step on the memory locations used to store the original matrix. In other words, if there is enough memory to store the full matrix in the first place, there is enough memory to invert it.

The disadvantages of direct solvers are their failure to exploit sparsity to reduce memory requirements and poor scaling of solution time with the number of degrees of freedom. For full matrices, the storage requirement is $O(N^2)$ and the solution time is $O(N^3)$, where N is the number of degrees of freedom. For banded matrices arising from three-dimensional problems, the storage requirement is $O(N^{5/3})$ and the solution time is $O(N^{7/3})$, based on an $O(N^{2/3})$ bandwidth for a mapped rectilinear grid. Banded methods also have an associated overhead for bandwidth minimization, usually paid in the mesh generation step.

It is also possible to write a direct solver that takes advantage of matrix sparseness but fills in some of the zero terms during the decomposition step [9]. An essential requirement for the computational efficiency of this scheme is a renumbering of the rows and columns to minimize fill-in. However, it has been our experience that even with the nested dissection renumbering scheme in [9], the fill-in for three-dimensional problems of practical size (tens or hundreds of thousands of degrees of freedom) is too large for the decomposed matrix to be stored in memory. For two-dimensional problems, the fill-in problem is less severe and the nested dissection solution method is generally

superior to the iterative methods to be discussed below, especially for solving multiple right-hand sides, as long as there is sufficient memory to store the fill-in terms.

Iterative methods attempt to decrease the exponent of N for both storage and solution time. If the matrix is completely sparse and the data structure used to store the matrix can exploit the sparsity, the storage for the problem reduces to $O(N^1)$ for any number of dimensions, because each row has on average the same number of non-zero coefficients no matter how large the problem grows. If the matrix is hybrid sparse/dense because of connections to an exterior "infinite" element, then in three dimensions the storage is $O(N^{4/3})$ assuming $O(N^{2/3})$ nodes on the exterior surface. As the number of degrees of freedom in a hybrid problem increases, a greater fraction of the total matrix storage is devoted to the exterior coupling.

Of the many iterative schemes, the most successful for a wide range of low-frequency electromagnetic problems has been the pre-conditioned conjugate gradient method, or PCCG. Of the pre-conditioning schemes, the most successful has been incomplete Cholesky decomposition, which preserves the sparsity structure of the original matrix equation [10]. Solution times have been observed in the range $O(N^{1.5})$ to $O(N^{1.8})$ on single-CPU machines. We have tested up to 300,000 nodes in a three-dimensional magnetostatic analysis and found matrix solution times scaling as $O(N^{1.56})$.

One of the key steps of the incomplete Cholesky pre-conditioned conjugate gradient (ICCG) algorithm is a *sparse* forward elimination defined as

$$x_i = \frac{x_i - \sum_{k=1+d_{i-1}}^{-1+d_i} L_k x_{j(k)}}{L_{d_i}} \tag{57}$$

where the sparse lower-triangular matrix, $[L]$, has been stored in a condensed form by scanning across rows, jumping over zeros. The diagonal entries in L are marked by d_i for row i, and $j(k)$ gives the column number of entry k in the sparse matrix. (57) is typical of the types of sparse matrix operations that must be performed in order to execute ICCG. We will delve more deeply into this subject in the next section.

Post-processing in electromagnetic finite elements is a grab-bag of calculations. Most are relatively quick and have little need for parallel processing. However, there are two exceptions: graphics display and

locating a point in the grid. Parallel processing for graphics displays, particularly three-dimensional displays, is such an active area for both research and product development that we cannot do it justice in this chapter. Despite the fact that high-performance, affordable graphics devices are an enabling technology for three-dimensional analysis — How else can one comprehend the millions of numbers in the output data? — we will concentrate here on the problem of finding the location of a point in a finite element mesh.

As discussed in the previous section, finite element analysis is based on the assumption that the unknown function(s) will be approximated by a set of interpolatory basis functions. The purpose of assembling and then solving the finite element matrix equation is to find the coefficients of the basis functions that give the best solution for a given mesh. (Some solvers also refine the mesh as required to achieve a defined level of error.) Once the coefficients are known, the user should be able to find the values of electromagnetic quantities of interest at any point within the problem domain, not just at finite element nodes. In order to accomplish this, it is necessary for the post-processing program to be able to take the coordinates of any point as input, and return not only the element it is in, but also its local coordinates within the element. With this information, the program can use the basis functions in the element to calculate the interpolated values of the desired quantities. Fig. 3.5 illustrates the problem of finding the corresponding element number (i) and local coordinates (u_1 and u_2) of a point given only its global coordinates (x and y) in a two-dimensional grid.

There are two phases to the location of a point in the finite element mesh. The first consists of running through the element list and determining whether each element deserves a closer look. For example, if the point is well outside a bounding box surrounding the element, there is no need to make a more accurate check. Another possibility is to use a tree structure for the mesh and find the element neighborhood of the point by recursive descent. Finally, if many points need to be found for plotting or numerical integration, it often saves time to check first the element matched for the previous point, on the theory that the points are probably close together.

Once an element has passed the first checks, it is necessary to find the local coordinates of the point by an iterative technique:

1. Guess a set of local coordinates for the point, like (0,0,0), that is a valid set for the element type (i.e. inside).

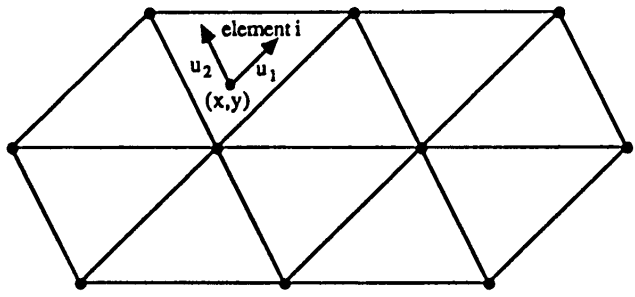


Figure 3.5 Location of a point in the finite element mesh.

2. Apply (32)–(34) to find the actual global coordinates of the point. Compute the difference between the target and actual points, Δ. If Δ is smaller than a specified tolerance, go to step 6. If too many iterations have passed, go to step 8.
3. Calculate the Jacobian matrix by (35)–(38) and invert it.
4. Compute the change in local coordinates using the inverse Jacobian matrix:

$$\begin{bmatrix} \Delta u_1 \\ \Delta u_2 \\ \Delta u_3 \end{bmatrix} = \begin{bmatrix} \frac{\partial u_1}{\partial x} & \frac{\partial u_1}{\partial y} & \frac{\partial u_1}{\partial z} \\ \frac{\partial u_2}{\partial x} & \frac{\partial u_2}{\partial y} & \frac{\partial u_2}{\partial z} \\ \frac{\partial u_3}{\partial x} & \frac{\partial u_3}{\partial y} & \frac{\partial u_3}{\partial z} \end{bmatrix} \begin{bmatrix} \Delta x \\ \Delta y \\ \Delta z \end{bmatrix} \tag{58}$$

5. Go to step 2.
6. Check the local coordinates. If they are within the element, go to step 7. Otherwise, go to step 8.
7. Terminate and report success.
8. Terminate and report failure.

The process of locating one point in the grid is reasonably fast, but it becomes CPU-intensive when many points need to be located in a large grid, particularly for three-dimensional post-processing. Parallel

processing could potentially speed up the algorithm by testing many elements simultaneously.

It is not strictly necessary to construct the finite element matrix completely, store it in one location, and then solve it. An alternate method would be to divide the entire finite element mesh into subregions. In each subregion, the degrees of freedom on the interior nodes can be eliminated in favor of the degrees of freedom on the boundary.

Consider the case of two subregions, 1 and 2, and the boundary nodes, b. The matrix equation can be divided as follows:

$$\begin{bmatrix} S_{11} & S_{1b} & 0 \\ S_{b1} & S_{bb} & S_{b2} \\ 0 & S_{2b} & S_{22} \end{bmatrix} \begin{bmatrix} X_1 \\ X_b \\ X_2 \end{bmatrix} = \begin{bmatrix} Y_1 \\ Y_b \\ Y_2 \end{bmatrix} \tag{59}$$

The submatrices $[S_{11}]$, $[S_{1b}]$, and $[S_{b1}]$ are calculated from elements within region 1 only, while $[S_{22}]$, $[S_{2b}]$, and $[S_{b2}]$ are calculated from elements within region 2 only. $[S_{bb}]$ has contributions from both regions. The interior variables are eliminated by inverting $[S_{11}]$ and $[S_{22}]$:

$$X_1 = [S_{11}]^{-1}Y_1 - [S_{11}]^{-1}[S_{1b}]X_b \tag{60}$$

$$X_2 = [S_{22}]^{-1}Y_2 - [S_{22}]^{-1}[S_{2b}]X_b \tag{61}$$

This leaves a reduced equation for the boundary variables only:

$$\begin{aligned} &([S_{bb}] - [S_{b1}][S_{11}]^{-1}[S_{1b}] - [S_{b2}][S_{22}]^{-1}[S_{2b}])X_b = \\ &Y_b - [S_{b1}][S_{11}]^{-1}Y_1 - [S_{b2}][S_{22}]^{-1}Y_2 \end{aligned} \tag{62}$$

The subregion formulation is not particularly efficient on single-CPU machines because it destroys the sparsity of the stiffness matrix. However, it is an interesting formulation for multiple-CPU machines, as we will see in the next section.

This completes our discussion of finite element paradigms. We will now consider the computing hardware that we can use to implement these paradigms.

3.4 High-Performance Computing Architectures

The discussion so far in this chapter has only touched peripherally on the subject of the different kinds of high-performance computing systems available to solve low-frequency electromagnetic problems. In this section, we will focus on the various high-performance computing architectures available to the numerical analysis community. We have deliberately avoided use of the term *supercomputer* because we do not wish to limit our investigation necessarily to only those systems with vector accelerators and/or multiple CPUs, which have traditionally been associated with supercomputers. These items may, in fact, be required to get the performance we need to solve large three-dimensional problems, but as users we are interested only in the final performance figures and how easy or difficult our algorithms are to implement on a particular machine.

Before discussing the high-performance architectures, one must have a conceptual framework for the classification of computing architectures [11,12,13]. One of the first schemes, and the most widely known, was suggested by Flynn in 1966 [14]. It characterizes an architecture by separate specification of single (S) or multiple (M) operations in its instruction stream (I) and data stream (D) respectively, resulting in a four-letter acronym.

The predominant type of computer until recently has been SISD: Single Instruction, Single Data or von Neumann architecture. The CPU progresses in an orderly manner from one instruction to the next, performing operations on data items one at a time. All widely used computer programming languages, particularly FORTRAN, were developed for SISD machines. Despite the limitations in the SISD approach, its great advantages are simplicity and familiarity — nearly all programmers learn to program SISD computers first, and then progress to more complex types as their skills improve. Fig. 3.6 is a schematic diagram of SISD operations. Note that our definition of SISD refers to the programmer's point of view and not necessarily the hardware. We follow the convention of including "architectures incorporating only low-level parallel mechanisms that have become commonplace features of modern computers" [13] in our definition of SISD. These low-level parallel features, such as simultaneous address calculation and floating-point arithmetic, are hidden from the programmer's view and will not affect our implementation of finite element methods.

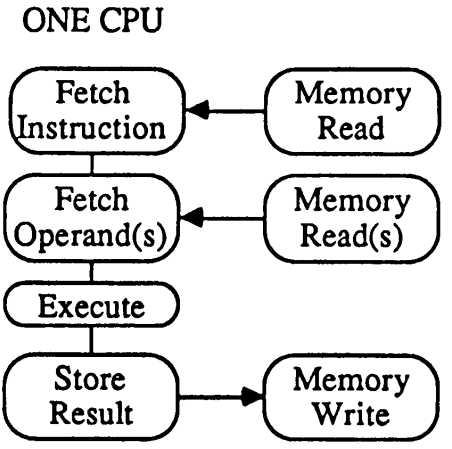


Figure 3.6 SISD architecture.

SIMD machines — Single Instruction, Multiple Data — were the first parallel architectures to attempt to break the "von Neumann bottleneck" by processing many data items simultaneously. The developers of the first SIMD machines recognized that in many scientific and engineering programs, the most ubiquitous and time-consuming operations are repeated additions, subtractions, and multiplications performed on arrays of numbers. If the order of the operations is not important, i.e. the calculation at a given spot in the array does not rely on the completion of the calculation at a previous spot, then the SIMD machine can set up one or more pipelines from memory to the arithmetic processors and back again, running the operations as fast as memory accesses and processing will allow. This is known as vector or array processing. Fig. 3.7 shows the operation of a SIMD-type vector processor.

Several other designs of more recent origin come under the SIMD category as well. For example, Thinking Machines Corporation's Connection Machine contains up to $65,536$ (2^{16}) single-bit processors under single-instruction control. Another recent SIMD machine is the Warp systolic array processor, developed by Professor H. T. Kung at

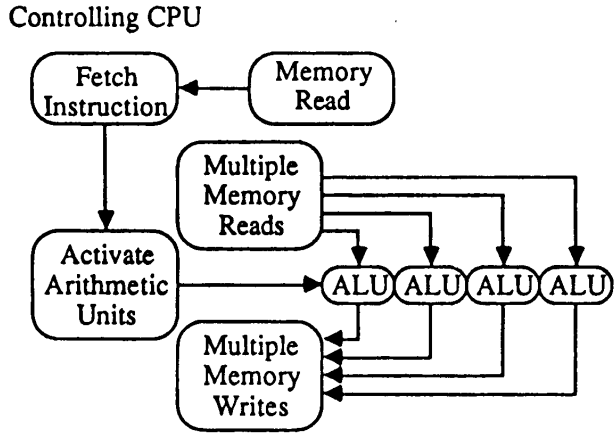


Figure 3.7 SIMD architecture.

Carnegie-Mellon University and produced commercially by Intel Corporation [15]. The Warp operates on a data stream by passing it from processor to processor in a pipelined manner without intermediate memory accesses. Each processor in the Warp is controlled by the same instruction stream, which is also passed from processor to processor so that the instructions are synchronized with the arrival of data. Of the commercially available high-performance computers, the Connection Machine and Warp are the most challenging for the implementation of finite element calculations because they depart so radically from the standard SISD computational model.

SIMD machines execute efficiently on the types of problems for which they were designed, but there are limitations to their application. Disappointed numerical analysts discovered soon after the first SIMD machines were programmed that the projected speed improvements were not obtained. The cause was found to be program sections that are not vector operations. In the limit that vector operations take zero time, the maximum throughput is determined by the execution of the non-vector (scalar) instructions in SISD mode. For example, if 10% of the execution time of a program in SISD mode is spent in scalar operations, then vector processing can provide at most a 10:1 speed improvement. The actual performance improvement will be less than

that, of course, since vector operations carry significant overhead and run at a finite speed. The formula expressing this behavior is called Amdahl's Law.

If f is the fraction of operations that must be performed sequentially, and the remaining operations are spread over P processors, then the simplest model of the resulting execution time, T, speedup, S, and efficiency, E, is [16]

$$T(P) = fT(1) + (1-f)\frac{T(1)}{P} \tag{63}$$

$$S(P) = \frac{T(1)}{T(P)} = \frac{P}{fP + (1-f)} \tag{64}$$

$$E(P) = \frac{S(P)}{P} = \frac{1}{1 + f(P-1)} \tag{65}$$

MIMD architectures — Multiple Instruction, Multiple Data — attack the problem of improving performance at a higher level than SIMD by providing multiple complete processors, each able to execute its own instruction stream independently. The basic idea is to divide the total problem into smaller sub-problems, each of which is done on a separate processor. Theoretically, this flexibility allows the application programmer to run a greater fraction of the problem in parallel than with vector processing alone. (Note that for MIMD machines the factor, f, in the above equations includes the overhead for synchronization and communication among the processors.) In practice, programming MIMD machines is difficult because of the inherent complexity of multiple processors doing different things simultaneously, as well as the general lack of automatic parallel decomposition software tools. In contrast, most SIMD (vector) FORTRAN compilers recognize loops that can be vectorized. Automatic parallel decomposition is an active research topic, and so hopefully MIMD machines of the future will be easier to program.

MIMD architectures divide into two broad classes: shared memory and distributed memory. Shared memory machines, as the name implies, have two or more CPUs with shared access to a large amount of common memory. Each CPU may or may not also have its own private memory. Interprocess communication consists simply of writing to and reading from the common data area, a fast operation because it occurs at random memory access rates. Shared memory architectures also lend themselves to conventional time-sharing applications with

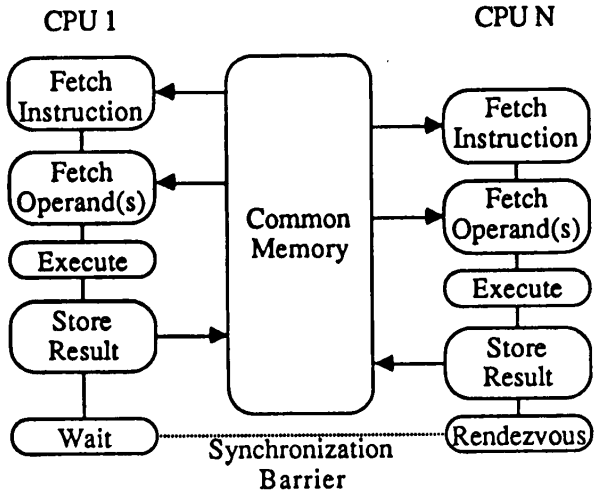


Figure 3.8 MIMD shared-memory architecture.

multiple users running multiple jobs. Since each user's program and data reside in a common memory area, jobs can be assigned to CPUs dynamically for the purpose of load balancing. Fig. 3.8 illustrates the operation of a shared-memory MIMD computer.

The principal limitation of shared memory machines is the bandwidth of the memory bus. When the bus is running at maximum speed, transferring the maximum amount of data each cycle (usually 32 or 64 bits), the bus is said to be saturated. When the memory bus is saturated, adding additional CPUs does not improve system throughput. In fact, more CPUs can actually decrease throughput due to the overhead of resolving bus contention. For this reason, most shared memory computers have a limit of 4–16 CPUs.

With the mass-production of 32-bit microprocessors and floating point accelerators, it has become feasible to build MIMD machines with hundreds or even thousands of powerful, general-purpose CPUs that are self-contained microcomputers. With so many CPUs, access to a common memory bus is out of the question. CPUs must communicate with other CPUs, but the communication paths must be limited to "nearest neighbors" in some sense. This type of MIMD computer is termed a distributed-memory, message-passing (DMMP) machine.

One method of connecting the communication paths among large

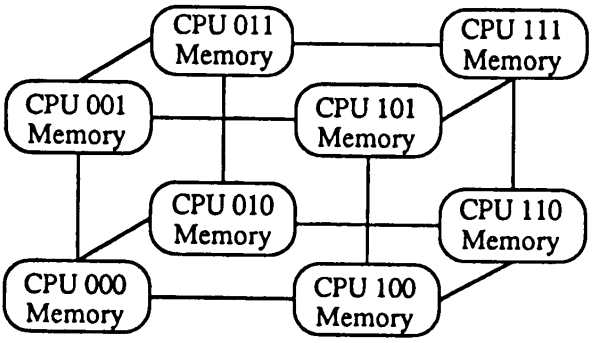


Figure 3.9 MIMD hypercube architecture (N = 3).

numbers of processors is the *hypercube* architecture [17]. Each CPU is a vertex on an N-dimensional cube, connected to each of its N nearest neighbors. There are 2^N CPUs and a total of ($N \cdot 2^N$) connections, counting input and output separately on each channel. Often there is also a local area network (Ethernet) connected to all CPUs for sending broadcast messages, such as the program and initial data. Fig. 3.9 shows the interconnections of a three-dimensional hypercube. Other methods of interconnection include rings, meshes, and trees [13].

Writing programs for distributed memory MIMD machines such as the iPSC Hypercube is more difficult than for shared memory MIMD machines because one must explicitly divide the data among the CPUs. Communication between CPUs, even for nearest neighbors, is an expensive overhead to be avoided. Since I/O (input / output) can be performed by only one processor in the hypercube, the final answers must be routed back over the interprocessor communication channels, often adding a considerable overhead even though the problem is nominally completed.[6] Despite the difficulties, distributed memory MIMD archi-

[6] Recent advances in hypercube computers have reduced these problems by providing faster communications, dedicated I/O processors on each node, and direct disk I/O from each node or small groups of nodes.

tectures have been successful for some classes of problems, as evidenced by the recent announcement of speedups of over 1000 for problems in fluid flow and stress analysis [18].

Although MIMD machines are capable of running much more than simple vector operations in parallel, the problem of synchronization adds a new potential for loss of efficiency. Consider, for example, a matrix solver in which the forward elimination step is performed in parallel on multiple CPUs. For simplicity, we will assume a full lower triangular matrix, L, although the same general method applies to sparse matrices as well using (57). Forward elimination proceeds as follows:

$$x_i = \frac{x_i - \sum_{j=1}^{i-1} L_{ij} x_j}{L_{ii}} \tag{66}$$

The synchronization problem lies in the fact that the calculation for x_i depends on the previous results for $x_j, j < i$. On SISD machines, this is not a problem because one simply arranges the algorithm so that the previous terms have already been calculated. On SIMD vector machines, the summation in (66) is done as a vector instruction, but again the previous terms have already been computed. However, in a MIMD implementation of forward elimination, there is no guarantee for an arbitrary CPU that the results from previous rows are complete. If the CPUs are progressing through the matrix in an orderly manner, then it is likely that the result from a previous row is ready when it is needed; but if it is not, a mechanism must be provided to have a CPU wait for another CPU to complete the required row. Depending on the efficiency of the synchronization mechanism, a large overhead will be introduced. We will see the effects of synchronization problems when we discuss the application of parallel processing to finite element analysis in the next section.

The SISD / SIMD / MIMD classification is a useful tool for understanding parallel architectures, but real parallel computers are more complicated than the simple models we have discussed. For example, many MIMD computers have CPU elements that can be considered to be SIMD vector processors in their own right, thereby combining the best of both architectures. However, a detailed investigation of the nuances of parallel architectures is beyond the scope of this chapter. The interested reader will find [13] a reasonable starting point for more information about modern computer architectures.

3.5 Mapping Finite Element Paradigms to Computing Architectures

We have come to the difficult task of matching the finite element computational paradigms to high-performance computing architectures. To repeat our comment from the previous section, our point of departure for all the paradigms is their implementation on SISD architectures. It is also our standard of performance comparison. In this regard, we should mention that the performance speedup measurement in (64) does not tell the complete story. In order to make efficient use of parallel processing, it may be necessary to use an algorithm that is not optimal for one processor but that scales well as many processors, P, are used in parallel. The speedup factor, $S(P)$, relates to the relative performance of one algorithm as the number of processors is increased. However, we should also consider the relative performance of the optimal algorithm (or the best one we can find) for a single processor versus another algorithm for solving the same problem that is optimized for many processors. In this way, we arrive at a more useful measurement of the benefits of parallel processing versus cost for a given problem. We will give a specific example of this below.

Of the four paradigms of finite element analysis — pre-processing (mesh generation), matrix assembly, matrix solution, and post-processing — we stand to gain the most per unit of programming effort by adapting the matrix solution step to high-performance computers. Once the matrix is assembled and stored in its sparse form, the computer programming required to implement the ICCG algorithm, for example, takes only a few hundred lines of FORTRAN code; the data structures and access methods tend to be more straightforward in matrix solvers than in the other parts of the finite element software package. Despite the relative simplicity and conciseness of the matrix solution subroutines, this step normally consumes at least half of the CPU cycles of the finite element analysis (not counting automatic mesh generation, if used). Furthermore, there is no I/O to perform during the matrix solution step, with the possible exception of virtual memory access that is handled by the operating system without explicit programmer intervention.[7] For these reasons, we will devote the bulk

[7] Out-of-core solvers, that store the matrix on disk and bring parts of it into memory as needed, have not proven efficient for sparse matrix operations. Unlike full or banded matrix solvers, in which there are

of this section to a discussion of how sparse matrix solution techniques map to several high-performance computing systems. First, however, we will mention briefly the possibilities that exist for speeding the execution of the other steps as well.

Vector operations on traditional SIMD computers are well suited to running inner loops in parallel, where long vectors are summed, scaled, etc. These operations occur throughout finite element programs, in all phases of the finite element analysis. Performing numerical analysis without vector capability means throwing away some of the most easily (and automatically) achieved parallel speedups. The major caveat with vector operations is that the overhead for startup on many machines is high enough so that vectorizing short loops is counterproductive, such as the inversion of the 3×3 Jacobian matrix in three-dimensional element stiffness matrix generation. Where to look for potential vector operations is a matter of detail and programming experience.

Having MIMD capability expands the horizons of parallel processing. The first obvious application is in matrix assembly. It makes sense to generate the element stiffness matrices in parallel, and then sum them into the global stiffness matrix. However, this implies the existence of an efficient mechanism for resolving access conflicts when summing the element matrices into the global matrix. Unless there is a reasonably efficient method of preventing two or more processes from summing to the same spot simultaneously, much of the parallel speedup will be lost. The best method would be a hardware *fetch and add* instruction that accumulates a sum in a common memory location with minimal synchronization overhead [19].

In the area of mesh generation, MIMD has the potential to speed up recursive subdivision techniques. Parallel recursive mesh generation would proceed along the following lines:

1. If the region is small or simple enough, generate finite elements immediately and return.

---

many operations to perform on each piece of the matrix as it is retrieved from memory, in sparse solvers each non-zero element of the matrix participates in only a few operations before the next piece of the matrix is needed. It has also been our experience that virtual memory is helpful only to a point. Physical random-access memory — including "solid-state disks" available on some supercomputers — should be sufficient to hold the matrix; otherwise, performance suffers considerably.

2. Split the region into N subregions.
3. For each subregion, check if another processor is free. If so, pass it one of the subregions to mesh.
4. For any subregions not passed to another process, call step 1 recursively.

This algorithm dynamically loads the processors depending on the evolution of the mesh splitting. No processor is idle for long. In order to work efficiently, it is essential that the algorithm be able to mesh each subregion without interfering with its neighbors, and to pass subregion data among processors much more quickly than the time to mesh subregions. Otherwise, synchronization overhead increases the sequential fraction of CPU time, f in (63)–(65) above, and destroys the efficiency of the procedure. It is also necessary for the supporting system software to allow recursive parallel calls, a feature that is difficult to implement.

Parallel processing to speed up non-graphical post-processing calculations proceeds along much the same lines as parallel element matrix generation. The set of N points can be split into M subsets, where M is the number of processors, and the element locations of each set of points can be found simultaneously and saved. Other post-processing calculations for multiple points can also be done in the same way.

Perhaps the most challenging match of finite element paradigm to parallel paradigm is mapping matrix assembly and solution to a distributed-memory machine, such as the Hypercube in Fig. 3.9. The alternative formulation in (59)–(62) has the potential to keep communication overhead low. If the mesh can be constructed so that processors assigned to neighboring mesh regions are also neighbors in the hypercube (a non-trivial task for a general, unmapped finite element mesh), then each processor in the hypercube can construct and invert its own local matrix for the interior nodes [20]. Boundary nodes can be handled in a number of ways; relaxation and other iterative schemes have the advantage of potentially running asynchronously, i.e. when a new estimate is ready for a boundary node, it can be communicated to adjacent processors as required and used from then on, without regard for the states of the other processors.

Unfortunately, this method is far from optimal for a one-CPU system. For example, consider the case of a two-dimensional mechanical finite element analysis for linear elastic uniform extension as discussed in [20]. The authors investigated partitioning this problem onto an Intel iPSC Hypercube (the original model with 80286-type processors)

with one, two, four, eight, and sixteen processors. To compare their parallel algorithm with an optimized scalar algorithm, we ran our own in-house two-dimensional stress analysis code on a Digital Equipment microVAX-II workstation, a single-user computer of the same vintage as the original iPSC Hypercube.

The first case in [20] involved a two-dimensional finite element mesh with 289 nodes and 542 degrees of freedom. This case was run on Hypercubes of one to sixteen processors. Our mesh was not exactly the same size — 280 nodes and 539 degrees of freedom — because of the different element types used — nine-noded quadrilaterals in [20] versus eight-noded quadrilaterals in ours. Table 3.1 summarizes our comparison.

Computer	CPU Time (Seconds)
iPSC — 1 processor	197
iPSC — 2 processors	175
iPSC — 4 processors	96
iPSC — 8 processors	56
iPSC — 16 processors	60
microVAX-II	55

Table 3.1. Timing for smaller mechanical FE model.

We also compared our program with the largest case in [20]: 4225 nodes and 8318 degrees of freedom (4256 nodes and 8435 degrees of freedom in our mesh). The authors of [20] only ran this case for sixteen processors because it was too big to run on fewer processors (insufficient total memory). Table 3.2 summarizes the comparison.

Computer	CPU Time (Seconds)
iPSC — 16 processors	2023
microVAX-II	1595

Table 3.2. Timing for larger mechanical FE model.

From these results, we conclude that we must be careful when evaluating the suitability of a particular parallel architecture for an application.[8] It is not enough to measure only the parallel speedup for

[8] These results should not be construed to detract in any way from the excellent research embodied in [20].

a particular algorithm; it is also necessary to compare the best parallel algorithm on a parallel computer with the best sequential algorithm on a uniprocessor computer. Only by making this comparison can we objectively measure the benefits of parallel processing.

We turn next to sparse matrix solution on high-performance computers, using the ICCG algorithm as an example. Sparse forward elimination (57) is typical of the sparse matrix operations we need to perform to implement ICCG. A simple FORTRAN subroutine to perform sparse forward elimination can be written as follows:

```
      SUBROUTINE SPRSFE(X,L,D,J,M)
      REAL X(*),L(*)
      INTEGER D(*),J(*),M
C     X HOLDS THE RIGHT-HAND SIDE
C     L HOLDS THE SPARSE LOWER-TRIANGULAR MATRIX
C     ---DIAGONAL TERMS STORED AS RECIPROCALS
C     D HOLDS THE DIAGONAL INDEX INTO L
C     J HOLDS THE COLUMN NUMBERS FOR L
C     M IS THE NUMBER OF DEGREES OF FREEDOM
      REAL SUM
      INTEGER I,K,KSTART,KEND
      DO 2 I=1,M
         SUM = X(I)
         IF (I.EQ.1) THEN
            KSTART = 1
         ELSE
            KSTART = 1+D(I-1)
         END IF
         KEND = D(I)-1
         DO 1 K=KSTART,KEND
            SUM = SUM-L(K)*X(J(K))
1        CONTINUE
         X(I) = SUM*L(D(I))
2     CONTINUE
      RETURN
      END
```

The diagonal terms of the lower triangular matrix were stored as their respective reciprocals because multiplication is a faster floating point operation on most computers than division.

The FORTRAN code above was not written with vector SIMD machines in mind. Even so, modern vectorizing compilers recognize the vector gather/scatter loop in DO 1 and insert the appropriate instructions. For example, the FORTRAN compiler on the Convex C240 gives the following message when directed to vectorize the SPRSFE subroutine:

Optimization by Loop for Routine SPRSFE

Line Num.	Iter. Var.	Reordering Transformation	Optimizing/Special Transformation
12	I		Scalar
20	K		FULL VECTOR Reduction

Line Num.	Iter. Var.	Analysis
12	I	Inner loop has induction value with varying base or step

The Convex C240 is a shared-memory MIMD machine with four processors. It can be programmed for parallel operation as well as vector operation. The Convex FORTRAN compiler can be directed to run programs in parallel as well as in vector mode. (Each CPU has its own vector hardware.) When directed to run in parallel, the Convex FORTRAN compiler gives the following diagnostic message, that differs from the previous one only by the assignment of multiple processors to work on the same inner loop:

Optimization by Loop for Routine SPRSFE

Line Num.	Iter. Var.	Reordering Transformation	Optimizing/Special Transformation
12	I	Scalar	
20	K	PARA/VECTOR	Reduction

The programmer convenience of this type of parallel decomposition is hard to beat, but in many cases we need to run an algorithm in

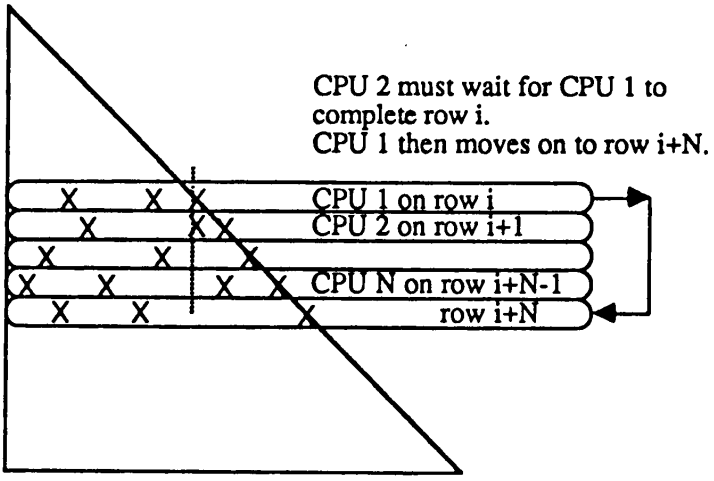


Figure 3.10 Parallel sparse forward elimination on a shared-memory MIMD computer.

parallel at a higher level. This is one of those cases because the DO 2 loop in SPRSFE runs over at most a few dozen non-zero matrix elements per row — enough to justify gather/scatter vectorization on one processor, but not enough work for multiple processors to share the same row, which the automatic parallelization on the Convex would do as indicated in the compiler's parallel optimization message above. We need to make the outer DO loop run in parallel instead of the inner one. This is possible, but only with modifications to the code. Assume we have M processors available. Each processor can be assigned to its own row of the matrix simultaneously, but provision must be made for synchronization waits if one of the elements of X from a previous row is required but it is not yet ready. Fig. 3.10 illustrates this requirement. (X marks non-zeros in the matrix.)

The following modification to SPRSFE handles the synchronization problem in a way that minimizes the delay time for the processors:

```
      SUBROUTINE SPRSFE(X,L,D,J,DONE,M,ID,N)
      REAL X(*),L(*)
      INTEGER D(*),J(*),M,ID,N
      LOGICAL DONE(*)
C     X HOLDS THE RIGHT-HAND SIDE
C     L HOLDS THE SPARSE LOWER-TRIANGULAR MATRIX
C     ---DIAGONAL TERMS STORED AS RECIPROCALS
C     D HOLDS THE DIAGONAL INDEX INTO L
C     J HOLDS THE COLUMN NUMBERS FOR L
C     DONE TELLS WHETHER ROW I IS READY
C     M IS THE NUMBER OF DEGREES OF FREEDOM
C     ID IS THE PROCESSOR ID OF THIS PROCESSOR,
C     --- 0 TO N-1
C     N IS THE NUMBER OF PROCESSORS
C     THESE VARIABLES ARE LOCAL TO THIS PROCESSOR:
      REAL SUM
      INTEGER I,K,KSTART,KEND
      DO 2 I=1+ID,M,N
         SUM = X(I)
         IF (I.EQ.1) THEN
            KSTART = 1
         ELSE
            KSTART = 1+D(I-1)
         END IF
         KEND = D(I)-1
         DO 1 K=KSTART,KEND
99          IF (.NOT.DONE(J(K))) GO TO 99
            SUM = SUM-L(K)*X(J(K))
1        CONTINUE
         X(I) = SUM*L(D(I))
         DONE(I) = .TRUE.
2     CONTINUE
      RETURN
      END
```

(Note that we assume no vector acceleration is available on the processors — statement 99 breaks the vector loop in DO 1.) The calling program uses a compiler directive (different for each vendor's system) to execute a parallel DO LOOP, calling the parallel version of SPRSFE for each processor ID:

```
CPAR$ DO-PARALLEL
      DO 100 ID=0,N-1
         CALL SPRSFE(X,L,D,J,DONE,M,ID,N)
100   CONTINUE
```

There is one major problem with this version of SPRSFE: the tight loop at statement 99 potentially wastes a great deal of CPU time doing nothing while waiting for another CPU to finish its row. The tradeoff is that the waiting CPU will be able to proceed almost immediately once the other CPU has finished its row. If the matrix is sparse, the number of synchronization loops executed should be small. Once the processors establish a natural phase delay at the beginning of the sparse forward elimination, they should proceed in an orderly way through the rest of the matrix with few delays. Given the sparsity structure or footprint of the matrix, this scheme guarantees that the process completes in the minimum elapsed time because the value of $X(J(K))$ is used as soon as it is ready if its unavailability has been blocking one or more processors.

On a single-user system, minimizing elapsed time for one job makes sense. However, few shared-memory multiprocessor systems are dedicated to one user. Instead, many users share one computer and the operating system allocates CPUs dynamically to active processes according to priority. An operating system capable of doing this is termed a symmetric multiprocessing operating system. A parallel job actually runs on multiple *processes* rather than *processors.* Each process is assigned a separate hardware CPU only when the CPUs are available in the priority scheme. For example, a parallel job with four processes running on a four-processor system will distribute over all four processors only when no other jobs are active. If the system is loaded, all four processes together may end up getting less than one equivalent CPU due to time-sharing with the other jobs.

Because of this feature of symmetric multiprocessing operating systems, running parallel jobs on busy shared-memory MIMD computers like the Convex C240 or Cray-2 is a counterproductive exercise. The number of floating point operations is the same as for a non-parallel

job; adding the overhead of starting and synchronizing the parallel loops, the total CPU time is more. If the operating system assigns priorities based on total CPU time used, a parallel job on a busy system often takes more elapsed time than a non-parallel job. When the system is lightly loaded, parallel processing becomes worthwhile because it allows one job to use otherwise idle CPU cycles, thereby decreasing the total turn-around time for the job.

In order to avoid wasting CPU cycles in tight loops that do nothing except wait for a flag from another process, symmetric multiprocessing operating systems allow the user to call a system service (again, different for each vendor) that checks the flag for a given amount of time. If the flag is not set by the end of the period (usually a few milliseconds), the operating system puts the process to sleep and checks again later. For example, statement 99 in the SPRSFE subroutine would be replaced by

```
99        CALL SPIN$WAIT(DONE(J(K)))
```

In our experience, this method can add even more overhead to the job than before because of the extra work required to put a process to sleep and wake it up again. Its only advantage over a tight loop is that it prevents one job from overwhelming the entire computer.

Another possibility exists for parallel processing of sparse forward elimination. So far, we have not assumed that the nodes are numbered in any particular order. If we were to use a banded solver, it would be necessary to renumber the nodes to minimize the bandwidth of the matrix. Suppose we were to do just the opposite: renumber the nodes to push the off-diagonal matrix elements away from the diagonal. If the off-diagonal elements are at least M away from the diagonal, then M rows can be processed in parallel without checking for completion. This idea can be applied to systolic arrays as well as MIMD shared-memory computers [21].

As a final, brief example of mapping finite element paradigms to computer hardware, consider the similarity between the Jacobian matrix transformations in (39) and (58) and the standard geometric display transformation:

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} r_{xx} & r_{xy} & r_{xz} & t_x \\ r_{yx} & r_{yy} & r_{yz} & t_y \\ r_{zx} & r_{zy} & r_{zz} & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \tag{67}$$

The three-dimensional Jacobian transformation is clearly a special case of the graphics display transformation when $t_x = t_y = t_z = 0$. Hardware instructions on graphics coprocessors, notably the Intel i860 [22], can speed this transformation significantly when the same matrix is applied to many vectors. Whether the speedup is worth the sacrifice in software portability is an open question.

3.6 Timing Comparisons for a Three-Dimensional Nonlinear Magnetostatic Finite Element Code

In this section, we will consider the performance of a complete finite element analysis code for three-dimensional nonlinear magnetostatic fields. This code, H3D, was developed by the author both for use in research and for applications in the design of electromechanical devices. It performs calculations typical of most finite element analysis programs, and so the relative performance numbers listed in this section should be similar for other finite element codes that use an in-memory matrix solver.

H3D uses the basic formulation expressed in (12)–(17) to solve for the reduced magnetic scalar potential, Φ_m. H3D handles the behavior of the far field by modeling outside the finite element grid with a harmonic series expansion. Degrees of freedom for nodes on the outer boundary are eliminated in favor of unknown coefficients in the harmonic series. The resulting finite element matrix footprint has the interesting structure shown in Fig. 3.11, assuming that the degrees of freedom for the harmonic coefficients are numbered first. The matrix is symmetric and positive-definite, and the upper left corner is full. Degrees of freedom for nodes just inside the outer boundary of the grid couple to all harmonic coefficients. However, note that Fig. 3.11 is not drawn to scale; in a typical problem there are only about one hundred degrees of freedom associated with harmonic coefficients but one hundred thousand degrees of freedom associated with nodal potentials. The upper left corner is therefore a small part of the total matrix.

We wrote a special matrix solver based on ICCG for H3D. Since the upper left corner and several rows (columns) are full, we could exploit vector processing on those matrix partitions without gather/scatter. We also provided the FORTRAN compilers on the VAX, Convex and Cray-2 computers with the necessary compiler directives to

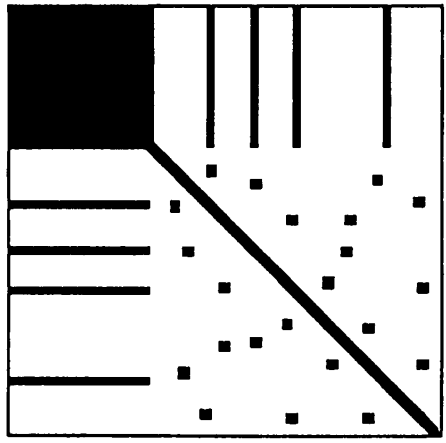


Figure 3.11 Matrix footprint for H3D

enable vector gather/scatter for all of the loops for the sparse part of the matrix. The pre-conditioner for the upper left part of the matrix is complete (i.e. exact) because that part of the matrix is full. Because of this feature, our hybrid dense/sparse ICCG solver scales extremely efficiently with problem size — $O(N^{1.56})$ where N is the number of nodes. Coupled with the $O(N^1)$ matrix storage requirement, this matrix solver has enabled us to handle problems with up to 300,000 nodes in a reasonable time (overnight) on a Convex C240 with 512 megabytes of memory.

We have not, however, attempted to create a parallel version of H3D. First, as discussed above, it does not seem to make much sense to use multiple processors on shared-memory MIMD machines like the Cray-2 or Convex C240 when many users are sharing the same systems. Programming DMMP computers for this problem appears to be a formidable task with a questionable payoff, as evidenced previously in this chapter for the mechanical finite element problem. For now, the SISD and vector SIMD (with gather/scatter) paradigms seem best suited to this application.

Table 3.3 summarizes the performance of the complete H3D code, counting disk I/O, for a small linear problem with 7,600 nodes as run

Computer	CPU Time (Seconds)
Tektronix XD88/30	799
DECstation 3100	668
VAX 6520 (1 CPU, no vector)	532
Sun SPARCstation-2	442
VAX 6520 (1 CPU + vector)	381
DECstation 5000/200	370
Convex C240 (1 CPU)	239
IBM RS/6000 Model 320	203
IBM RS/6000 Model 530	144
IBM RS/6000 Model 540	120
Cray-2 (1 CPU)	82

Table 3.3. H3D Timings (7,600 Nodes).

Computer	CPU	Elapsed
Sun SPARCstation-2	11,667	12,387
DECstation 5000/200	10,712	11,036
Convex C240 (1 CPU)	8,664	12,875
IBM RS/6000 Model 530	5,591	5,714
IBM RS/6000 Model 540	4,686	4,711
Cray-2 (1 CPU)	4,662	12,207

Table 3.4. H3D Timings (34,124 Nodes).

on several workstations[9] in addition to the Digital Equipment VAX 6520, Cray-2 and Convex C240. In Table 3.4, we compare the Cray-2 and Convex C240 with the IBM RS/6000, Sun SPARCstation-2, and DECstation 5000/200 workstations for both CPU and elapsed times on a larger, nonlinear case with 34,124 nodes.

[9] The Tektronix XD88/30 is a workstation based on the Motorola 88000 RISC CPU chip. The DEC3100 is a workstation based on the MIPS R2000 RISC CPU chip. The DEC5000 is a workstation based on the MIPS R3000 RISC CPU chip. The IBM RS/6000 workstation models use IBM's new RISC CPU chip with various clock speeds and cache sizes. The HP 9000 Model 720 uses Hewlett-Packard's proprietary Precision Architecture RISC chip. The Sun SPARCstation-2 uses the SPARC RISC chip.

Since H3D was written in FORTRAN, these timings reflect the efficiency of the machine code produced by the respective compilers as well as the raw CPU speed. The IBM compiler in particular is able to recognize parallelism at a low level and automatically schedule multiple operations simultaneously in the different processing sections of the CPU [24]. However, we made no attempt to tailor H3D for any particular computer other than the compiler directives for vectorization as mentioned above.

We leave it to the reader to draw his or her own conclusions regarding the numbers in Table 3.3 and 3.4, except to note that performance of the RISC-based workstations was still increasing at a rapid pace as this chapter was written.

3.7 Conclusion

Recall the key question from the introduction: If an algorithm does not map well to a particular hardware architecture, should we change the hardware or the algorithm? It is not an easy question to answer. Before attempting an answer, we will review the arguments of the chapter.

In low-frequency problems in electromagnetics, the electric and magnetic fields are either uncoupled (electrostatic and magnetostatic problems) or weakly coupled (eddy current problems). Because of this, Maxwell's divergence equations (2) and (4) do not follow automatically from the respective curl equations (3) and (1), but must be enforced explicitly. Straightforward manipulation of Maxwell's equations relevant to each problem type, including the divergence equations, leads to second-order partial differential equations (PDEs) (7) or (20).

Aside from a few canonical geometries, (7) and (20) must be solved numerically for cases of practical interest. Before considering a method, we must first decide on a numerical representation of the solution. The representation should have the following features:

1. Express the solution over all space with a finite number of variables.
2. Approximate any function — and its first derivatives — in the solution space of the PDE to arbitrary accuracy as the number of variables is increased.
3. Allow local refinement of the solution accuracy as needed.

4. Track the geometric shapes (boundaries) of the objects modeled to arbitrary accuracy as the number of variables is increased, including geometric features of different size scales.
5. Allow modeling of nonlinear material properties.

Finite elements satisfy all of these criteria. Other schemes, such as boundary elements and finite differences, fail to satisfy all the above criteria, although they remain useful in special cases.

Having settled on the finite element method for the above reasons, it is possible to establish an error measure (43) and to minimize the error by solving a matrix equation for the degrees of freedom in the finite element mesh: nodal potentials, nodal vectors, or edge vectors. The matrix equation is sparse as a direct result of the local support property of finite elements. Using storage techniques and equation solvers optimized for sparse matrices, we can solve large three-dimensional problems having hundreds of thousands of degrees of freedom.

Electromagnetic numerical analysis is a fascinating discipline for its practitioners, but it is the end users of the methods who ultimately give the field relevance and life. High-performance computers enable end users to model problems with increasing levels of detail and accuracy. A revolution is well under way that will ultimately replace expensive cut-and-try design methods, not only in electromagnetics but all other engineering disciplines as well. The prime requisites for this revolution to succeed are

1. Stable and accurate numerical methods that can be applied easily by non-experts to complex models (i.e. methods that satisfy criteria 1–5 at the beginning of this section),

and

2. Affordable computers with enough power to solve the complex models.

We take it as axiomatic that the first requisite should be met before the second is tackled. As computer consumers, we can select the hardware which best satisfies the second criterion once we have settled on a suitable analysis method. Does it make sense to select a high-performance architecture first, and then — like the legendary giant, Procrustes, who cut the legs of his visitors to fit his bed [25] — cut our analysis method to suit? We think not. Finite elements are a proven technique for solving low-frequency problems in electromagnetics; we are disinclined to switch to a different method simply because some of the finite element paradigms are difficult to implement on particular

high-performance computers.

At the same time, we should not ignore the possibilities that exotic types of computers provide us for delivering higher performance than traditional SISD and SIMD vector machines. DMMP machines, like the iPSC Hypercube, hold the greatest promise for increasing performance by scaling up the number of processors into the thousands, while being flexible enough to program complex applications. As processor and memory prices continue to fall, this architecture theoretically gives us a way to build computers with whatever level of performance we require for our applications.

Unfortunately, DMMP machines remain relatively expensive in configurations with enough processors and memory to be useful for large three-dimensional analysis problems. They also remain more restrictive in the types of algorithms that map efficiently and more difficult to program than SISD, vector SIMD, or shared-memory MIMD computers.

With many different high-performance architectures to select, it makes sense for us to stay with the basic finite element approach for low-frequency electromagnetic problems rather than switch to a new approach whenever a new type of hardware comes along. We will continue to favor those systems we can program quickly and reliably to execute the finite element paradigms. Software support and hardware reliability are just as important as theoretical peak Mflops ratings. While they remain interesting and valuable research tools, massively parallel SIMD and MIMD machines have not yet been able to deliver competitive price/performance to end users. Recently announced RISC workstations from Sun, Digital Equipment, International Business Machines, and Hewlett-Packard have brought three-dimensional analysis to the individual engineer's desktop. In many cases, these individual workstations give faster job turnaround than identical programs running on heavily loaded supercomputers. In the near term, MIMD shared-memory compute servers and workstations with one to four RISC processors are more likely to deliver affordable power to end users than massively parallel machines. There will always be a place for supercomputers for large, high-priority jobs; but for now, workstations deliver Mflops to individual engineers more cost-effectively.

3.8 Addendum

In the 2 years since the original manuscript for this chapter was written, significant developments in computer hardware and software have brought parallel processing within an affordable range for most engineering departments. On the hardware side, notable new systems include multiprocessor plug-in cards for PCs, the Sun SPARCstation-10 multiprocessing engineering workstation, and closely coupled networks of high-performance workstation servers from Hewlett-Packard and Digital Equipment. These systems have the potential to bring supercomputer-class power to engineering departments for under $100,000. However, lack of application software to take advantage of these parallel computers remains a barrier to their effective use.

Software development aids for these parallel computers are slowly emerging. While automatic parallel decomposition remains a long-term goal, significant help for the developer of parallel application programs can be found in vendor-specific parallel processing libraries from parallel computer vendors such as Intel and Thinking Machines. There are numerous commercial and public domain software tools to aid in vectorization, parallelization, and performance monitoring. New versions of the venerable FORTRAN language that offer explicit parallel language constructs are also just over the horizon.

Perhaps the most interesting development in software since the original manuscript for this chapter is a package called PVM - Parallel Virtual Machine - from Oak Ridge National Laboratories. This package can be freely downloaded over the Internet. From the application programmer's perspective, PVM is a set of subroutines callable from FORTRAN or C that handle message creation, transmission, and reception over a network of computers in much the same way that vendor-specific libraries work for MIMD/DMMP machines like the Intel Hypercube. PVM automatically converts data formats among computers from different vendors, making it possible to run an application in parallel over a collection of heterogeneous computers.

PVM is also becoming a de facto standard for the next generation of closely-coupled parallel computers. This raises the intriguing possibility of developing a parallel algorithm, coding it with PVM, running it on a network of workstations, and then finally running the identical source code on a true parallel computer (i.e., a computer designed for parallel processing with multiple processors connected via high-speed

data links rather than a standard local area network). PVM also works well in passing messages among multiple processes in a multiprocessor computer such as the SPARCstation-10, because messages can be communicated rapidly though shared memory. The actual hardware and low-level software protocols for data transfer are hidden from the application programmer by PVM. While we have found that writing programs under PVM is no easier than using vendor-specific libraries, once written the source code can be moved with no changes to almost any parallel environment that supports the message-passing paradigm.

The author and colleagues have recently rewritten the H3D computer program, previously discussed in this chapter, for parallel execution under PVM. One of the challenges of parallel programming for finite element calculations is keeping the ratio of computation to message passing high. This has been difficult in the past because the sparse matrix operations encountered in finite element analysis have a lower ratio of computational operations per memory storage location than do full matrix operations for methods such as integral equations. With each parallel process running on a full-blown workstation with virtual memory, we can partition the problem into larger blocks than we could on early versions of parallel computers such as the Hypercube. We have demonstrated effective CPU utilization of around 80% when running a sparse matrix solver over a lightly loaded Ethernet local area network. When we move on to faster communication schemes, such as FDDI or local bus interconnects, we expect to do even better. Further discussion of our efforts to parallelize the H3D code using PVM can be found in [26].

On the whole, the author's outlook for applying parallel processing to finite element methods is more optimistic now than two years ago. Procrustes finally decided to lengthen his bed rather than shorten his guests.

References

[1] Jackson, J. D., *Classical Electrodynamics*, New York: John Wiley & Sons, Inc., 1962.

[2] Konrad, A., "A differential equation formulation for radiation problems based on the continuity gauge instead of the Lorentz gauge," *Digest of the 1986 IEEE AP-S International Symposium on Antennas and Propagation,* Philadelphia, Pennsylvania, 915–918, June 8-13, 1986.

[3] Barton, M. L., and Z. J. Cendes, "New vector finite elements for three-dimensional magnetic fields computations," *Journal of Applied Physics,* **Vol. 61,** No. 8, 3913–3921, 1987.

[4] Bossavit, A., "A rationale for 'edge-elements' in 3-D fields computations," *IEEE Transactions on Magnetics,* **Vol. 24,** No. 1, 74–49, January 1988.

[5] Bedrosian, G., M. V. K. Chari, M. Shah, and G. Theodossiou, "Axiperiodic finite element analysis of generator end regions Part I — Theory," *IEEE Transactions on Magnetics,* **Vol. 25,** No. 4, 3067–3069, July 1989.

[6] Zienkiewicz, O. C., *The Finite Element Method in Engineering Science*, London: McGraw-Hill, 1971.

[7] *Webster's New Collegiate Dictionary*, Springfield, Massachusetts: G. & C. Merriam Company, 1973.

[8] Graichen, C. M., A. F. Hathaway, P. M. Finnigan, A. Kela, and W. J. Schroeder, "A 3-D fully automated geometry-based finite element meshing system," *Proceedings of the ASME 1989 Winter Meeting*, San Francisco, California, 1989.

[9] George, A., and J. W. H. Liu, *Computer Solution of Large Positive Definite Systems*, Englewood Cliffs, New Jersey: Prentice-Hall Series in Computational Mathematics, 1981.

[10] Manteuffel, T. A., "The shifted incomplete Cholesky factorization," SAND78-8226, Sandia Laboratories, May 1978.

[11] Agrawal D. P., (ed.), *Advanced Computer Architecture*, IEEE Computer Society Press, 1986.

[12] Händler, W., "The impact of classification schemes on computer architecture," *Proceedings of the 1977 International Conference on Parallel Processing*, 7–15. (Reprinted in [11], 3-11).

[13] Duncan, R., "A survey of parallel computer architectures," *Computer*, **Vol. 23**, No. 2, 5–16, February 1990.

[14] Flynn, M., "Very high computing systems," *Proceedings of the IEEE*, **Vol. 54**, 1901–1909, 1966.

[15] Kung, H. T., "Why systolic arrays?" *Computer*, 37–46, January 1982, (Reprinted in [11], 300-309).

[16] Jordan, H. F., "Experience with pipelined multiple instruction streams," *Proceedings of the IEEE*, **Vol. 72**, 113–123, 1984.

[17] Bhuyan, L. N., and D. P. Agrawal, "Generalized hypercube and hyperbus structures for a computer network," *IEEE Transactions on Computers*, **Vol. C-33**, No. 1, 323–333, April 1984. (Reprinted in [11], 207–217).

[18] "Sandia trio wins awards at Compcon," *The Institute, News Supplement to IEEE Spectrum*, **Vol. 12**, No. 5, 8 May 1988.

[19] Gottlieb, A., R. Grishman, C. P. Kruskal, K. P. McAuliffe, L. Rudolph, and M. Snir, "The NYU Ultracomputer — designing an MIMD shared memory parallel computer," *IEEE Transactions on Computers*, **Vol. C-32**, No. 2, 175–189, February 1983.

[20] Carter, W. T., T.-L. Sham, and K. H. Law, "A parallel finite element method and its prototype implementation on a hypercube," *Computers & Structures*, **Vol. 31**, No. 6, 921–934, 1989.

[21] Hammond, S. W., and G. Bedrosian, "Solution of large, sparse, linear systems on systolic arrays," General Electric Technical Information Series, 86CRD166, GE Corporate Research & Development, Schenectady, New York, October 1986.

[22] Grimes, J., L. Kohn, and R. Bharadhwaj, "The Intel i860 64-bit processor: a general-purpose CPU with 3D graphics capabilities," *IEEE Computer Graphics and Applications*, **Vol. 9**, No. 4, 85–94, July 1989.

[23] Chari, M. V. K., and P. P. Silvester (ed.), *Finite Elements in Electrical and Magnetic Field Problems*, New York: John Wiley & Sons, 1984.

[24] Misra, M., (ed.), *IBM RISC System/6000 Technology*, Austin: IBM Austin Communications Department, 1990.

[25] Bulfinch, T., and E. Blaisdell, *Mythology: The Age of Fable, The Age of Chivalry, Legends of Charlemagne*, New York: T.Y. Crowell Co., 1947, p. 124. (Popularly known as "Bulfinch's Mythology.")

[26] Bedrosian, G., and R. W. Benway, "Magnetostatic finite-element analysis on MIMD/DMMP parallel computers," *Journal of Applied Physics*, **Vol. 73** No. 10, 6772–67777, 15 May 1993.