Chapter 2

Software and Hardware Models

In this chapter, PDEs, solution methods, and the underlying structure of many large-scale PDE applications used in science and engineering are introduced and analyzed. Next, the exploitation of parallelism to reduce solution times and allow increased model complexity is discussed. A methodology for parallelizing PDE applications using an intuitive coarse-grain approach for exposing the underlying parallelism of the solution methods is presented. Once the software model has been defined, a brief overview of relevant parallel architectures is given. Finally, given both the hardware and software background, the portability of resulting parallel PDE applications is argued.

2.1 PDEs and Approximate Solutions

The main focus of this chapter is on the computer software and hardware for solving PDEs; however, before discussing the construction of PDE applications and their requisite computational modules in-depth, a general discussion of systems of PDEs is in order. By examining the breakdown of a set of continuous equations into small, discrete, and soluble components, the framework of the resulting computer applications becomes clear. Examples of PDE systems include the equations arising in the study of CFD which are discussed in Shakib's thesis [48] and the equations for structural mechanics which are discussed in Ferencz's thesis [16]. Another PDE family of wide interest is that used for semiconductor device characterization. One popular form will be used as a concrete example in the remaining discussions. PDEs in other application domains may be treated in a similar fashion.

Charge transport in semiconductor devices is best described by the Boltzmann transport equation (BTE). For a thorough discussion of the various computer methods for the solution of the BTE, see Aluru's thesis [1]. One popular approximation to the BTE is the drift-diffusion model [50]. The set of equations used by the drift-diffusion model can be written as shown below. Poisson's equation (2.1) describes the electrostatic potential and the continuity equations describe carrier concentrations for electrons (2.2) and holes (2.3):

$$\nabla(\varepsilon\nabla\Psi) = -q(p-n+N_D^+-N_A^-)-\rho_F \tag{2.1}$$

$$\frac{\partial n}{\partial t} = \frac{1}{q}\nabla\bullet J_n - U_n \tag{2.2}$$

$$\frac{\partial p}{\partial t} = \frac{1}{q}\nabla\bullet J_p - U_p \tag{2.3}$$

Ψ is the electric potential, and n and p are the electron and hole concentrations. N_D^+ and N_A^- are the ionized impurity concentrations and ρ_F is the fixed-charge density. U_n and U_p are the electron and hole recombination rates. J_n and J_p are the electron and hole current densities and are proportional to $-\nabla\Psi$ as shown below:

$$J_n = q\mu_n E_n n + qD_n\nabla n \tag{2.4}$$

$$J_p = q\mu_p E_p p - qD_p\nabla p \tag{2.5}$$

where

$$E_n = E_p = E = -\nabla\Psi \tag{2.6}$$

μ_n and μ_p are the electron and hole mobilities. D_n and D_p are the electron and hole diffusivities.

In general, these equations have no closed form solution; however, an approximate solution can be computed using an appropriate discretization method. That is, an approximate solution is obtained at a series of discrete points in the problem domain rather than obtaining a continuous solution to the equations. To facilitate this, the semiconductor device (also called the “domain”) to be characterized must be described geometrically. Next, an appropriate wire-frame mesh which overlays the simulation domain is created. Intersections of grid lines are referred to as “nodes” while the faces enclosed by a set of grid lines are called “elements.” To illustrate, the simulation grid for a complimentary metal-oxide-semiconductor (CMOS) inverter, a basic building block of many modern integrated circuits, is shown in Figure 2.1. The behavior of this device may be characterized by equations (2.1) - (2.6) above.

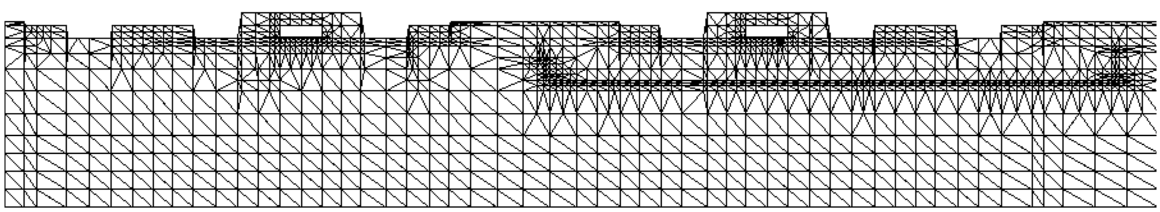

Figure 2.1 Grid structure of a CMOS inverter.

Once the grid-based (discrete) structure is defined, the continuous variables in the PDEs are replaced by vector values at the nodes of the simulation grid; the differential operators are replaced by suitable discretization operators. If more than one quantity is desired at a node, that node defines multiple degrees of freedom. For example, solving the three device equations simultaneously yields three degrees of freedom per node. Common discretization methods include the finite element (FE) method, the finite difference (FD) method, and the finite volume (FV) method. Once the domain has been discretized, an appropriate solution scheme may be applied to the resulting equations.

2.1.1 Discretization Methods

Most discretization methods seek to decompose problems into a collection of smaller, more easily understood and more easily soluble pieces. The behavior of each piece is generally governed by its local environment. The local neighborhood rarely consists of more than the element (or node) itself and possibly the adjoining elements (or nodes). The aggregate solution is then the sum of the local solutions. The following discussion of FD, FE, and FV methods is not intended as a thorough treatment of approximation methods but is intended to summarize the methods while highlighting data dependencies inherent in each method. (For a more complete discussion of FD, FE, and FV methods as they are applied to semiconductors, see Price [43] and Snowden [50].)

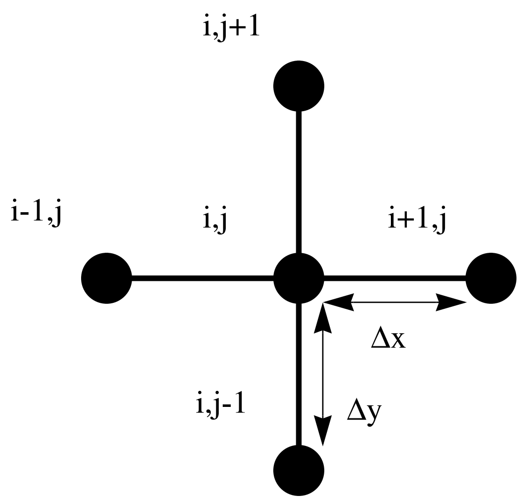

Figure 2.2 Five-point finite-difference stencil.

The FD method uses difference quotients at the nodes in place of the original partial derivatives. To illustrate, the classic 5-point finite difference stencil (for a uniform mesh) used in many 2-D applications is shown in Figure 2.2. In the classic 5-point stencil, the value at each node (i,j) depends on its own value and that of its four immediate neighbors. The finite-difference equation for the Poisson equation (2.1) using the 5-point stencil may be written as:

$$\frac{\Psi_{i-1,j} - 2\Psi_{i,j} + \Psi_{i+1,j}}{\Delta x^2} + \frac{\Psi_{i,j-1} - 2\Psi_{i,j} + \Psi_{i,j+1}}{\Delta y^2} = -\frac{q}{\varepsilon_0 \varepsilon_r}(N_{D_{i,j}} - n_{i,j} + p_{i,j} - N_{A_{i,j}}) \tag{2.7}$$

Applying (2.7) to each node in a simulation grid results in a system of equations for the value of Ψ at each mesh point. Due to the solely local dependencies for each (i,j), this system of equations exhibits an extremely sparse structure. The sparsity pattern arising from the two-dimensional 5-point discretization of Figure 2.2 is shown in Figure 2.3. In the case of non-uniform meshes or more complex stencils, the difference equation becomes more complicated but exhibits the same general form.

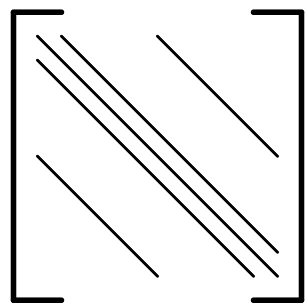

Figure 2.3 Sparsity structure for a five-point FD discretization.

The FE method was first developed to solve structural mechanics problems; later work has shown it to be a powerful and flexible method for solving many different types of PDEs. The FE method finds an approximate solution to a differential equation at each element in the simulation mesh. The solution over each element is assumed to follow a simple function. The elemental solutions are combined to form a global solution. Again using Poisson's equation (2.1) as an example, the FE method produces an approximate solution Ψ^e. The total approximate solution is given by the sum of the n elemental solutions:

$$\Psi^e = \sum_{j=1}^{n} \Psi_j^e \tag{2.8}$$

To insure a proper form for the solution at each element, a smooth polynomial function is used. "Shape functions" (Θ_i) are then defined for the number of degrees of freedom (m) in each element. The elemental approximation may then be written as:

$$\Psi_j^e = \sum_{i=1}^{m} \Psi_i \Theta_i(x, y) \tag{2.9}$$

Two-dimensional shape functions are defined for $1 \le i \le m$ as:

$$\Theta_i(x_j, y_j) = 0 \text{ for } i \ne j \tag{2.10}$$

$$\Theta_i(x_j, y_j) = 1 \text{ for } i = j \tag{2.11}$$

Furthermore, the shape functions are defined to be zero in all elements not containing node j. Many possible shape functions exist which meet these criteria.

The two-dimensional finite element approximation to Poisson's equation is obtained by multiplying Poisson's equation by Θ_i and integrating over the domain (Ω). The result, after simplification, is shown in Equation (2.12).

$$\sum_{j=i}^{m} [K_{ij}^{\Psi} \Psi_j^K - qM_{ij}(N_{Dj}^e - n_j^e + p_j^e - N_{Aj}^e)] = 0 \qquad 1 \le i \le m \tag{2.12}$$

where the stiffness matrix K is defined as:

$$K_{ij}^{\Psi} = \int_{\Omega} \varepsilon \nabla\Theta_i \bullet \nabla\Theta_j ds \tag{2.13}$$

and the mass matrix M is defined as:

$$M_{ij} = \int_{\Omega} \Theta_i \Theta_j ds \tag{2.14}$$

The nature of the shape functions insures that each node interacts with only a small number of other nodes in the resulting system. This results in a sparse matrix structure with N equations and N unknowns (where N is the number of degrees of freedom in the approximation). These systems of equations are more general and less structured than the sparsity pattern shown in Figure 2.3, but of the same type.

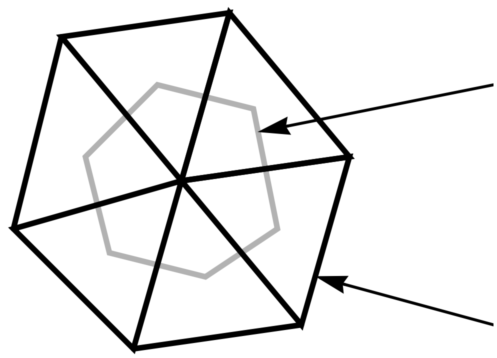

Figure 2.4 Sample finite-volume integration.

The FV method can be characterized as a hybrid of the FE and FD methods, allowing more flexibility in the handling irregular structures than FD with less cost than the FE approach. For Poisson's equation on an irregular triangular grid as in Figure 2.1, each equation is integrated over a small polygon enclosing each node (Figure 2.4). The integration equates the flux into the polygon with the sources and sinks inside it. Figure 2.5 shows a single element and the values needed to compute its contribution to node *i*. The FV approximation yields the following equation:

$$\sum_{1 \le m \le M} \varepsilon_m \vec{E}_m \cdot (h_{jm}\vec{u}_{jm} + h_{km}\vec{u}_{km}) = \rho_i A_i \qquad (2.15)$$

The vectors u_j and u_k are the unit vector in the *ij* and *ik* directions. The quantities h_j and h_k and the lengths of the boundaries segment of this portion of the enclosing polygon. M is the number of triangles comprising the bounding polygon (i.e., containing node *i*) and A_i is the area of the polygon enclosing node *i*.

The dot product $\vec{E} \cdot \vec{u}_j$ is the component in the *ij* direction. By assuming constant charge within the integration area, $\vec{E} \cdot \vec{u}_j$ can be written as:

$$\vec{E} \cdot \vec{u}_j = -\frac{\Psi_j - \Psi_i}{d_j} \tag{2.16}$$

where d_j is the distance between nodes i and j. Substituting (2.16) into (2.15) yields:

$$\sum_{1 \le m \le M} \varepsilon_m \left((\Psi_i - \Psi_{jm}) \frac{h_{jm}}{d_{jm}} + (\Psi_i - \Psi_{km}) \frac{h_{km}}{d_{km}} \right) = \rho_i A_i \tag{2.17}$$

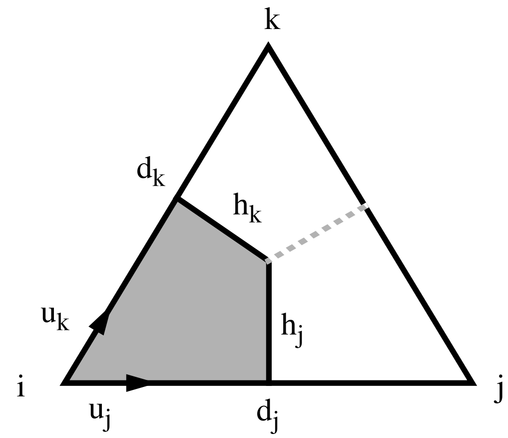

Figure 2.5 A single element in the discretization. Shaded area represents the contribution of this element to node *i*.

Performing this summation for each node in the simulation mesh creates a system of *N* equations with *N* unknowns. Once again, the system of algebraic equations exhibits a general sparse pattern. Equation (2.17) illustrates the local nature of the FV method - node *i* depends only upon its neighboring nodes.

2.2 Computer Software for the Solution of PDEs

As described above, the various methods for approximating a system of PDEs creates a computational surrogate for the original equations. It is important to note that the solution of the resulting system requires a great deal of work and was impractical until the advent of modern digital computers. Fortunately, these methods utilize algorithms easily translated into computer programs. A computer application to solve PDEs merely applies an approximation and then solves the resulting system of equations. A wide range of applications to solve the wide range of PDEs encountered in both science and engineering have been developed throughout the last thirty years. Despite the diverse solution methods employed to solve these diverse sets of PDEs, many PDE applications share a common global program structure. This commonality arises from the discrete, grid-based nature of most approximation schemes. The main difference lies with the various methods used to translate the continuous form of the PDEs into a discrete form which can then be solved by computer. The majority of the mathematical machinery subsequently employed to find a solution to the discretized systems (e.g., non-linear solvers and linear solvers) tend to be general purpose, allowing them to be employed in a wide range of PDE applications.

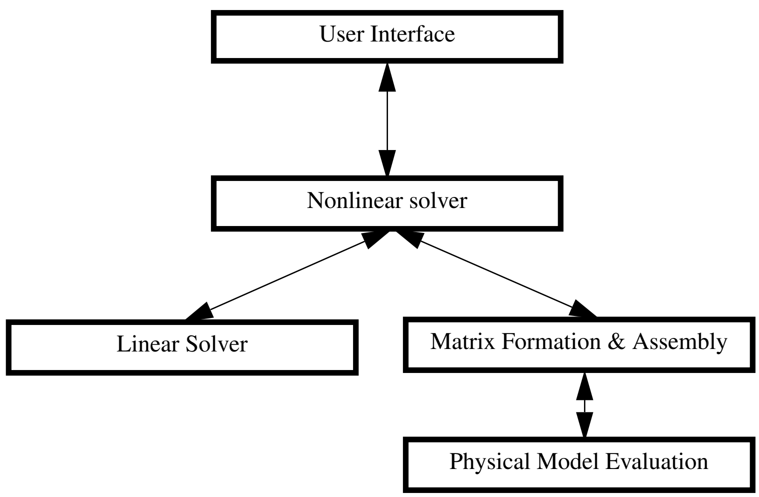

Figure 2.6 PDE-solver execution structure.

The coarse-grain structure of a typical grid-based nonlinear PDE solver is shown in Figure 2.6. This schema represents the main program components and the associated control hierarchy. The application consists of modules to perform each of the following tasks: user interface (UI), nonlinear solution, linear solution, matrix formation, and physical model evaluation. The UI provides an external interface for the application and facilitates user interaction with the solver modules. The nonlinear and linear solution modules contain the mathematical machinery used to solve the set (or sets) of equations arising from the PDE approximation scheme. Matrix formation and physical model evaluation are the two program components which employ an application's approximation scheme.

2.2.1 User Interface

As with most user-directed computer applications, a UI is required to interact with users. Typical tasks for a PDE solver UI might include the parsing of input, file I/O, and visualization of simulation results. Older codes tend to use simple command-line parsers supporting a spartan command language. Newer codes take advantages of advances in graphical user interface (GUI) techniques to simplify users' interactions with the solver. Regardless of its implementation, the UI is responsible for translating users' commands into the requisite sequence of low-level actions to be carried out by the solution modules. This is typically managed via a combination of setting appropriate option flags and calling the necessary functions. Notwithstanding its critical functionality, the UI usually performs little work (and requires negligible execution time) relative to the PDE solution.

2.2.2 Nonlinear and Linear Solutions

The approximation of PDEs gives rise to a set of algebraic equations. In many cases, they are nonlinear, requiring linearization and the repeated solution of the resulting linearized system. (Linear problems, such as those arising from structural mechanics, may be solved in a simpler fashion by a single linear step.) The linearized form of the PDE is then solved using a linear solution method. In the case of coupled sets of PDEs (e.g., Poisson and continuity), there are two common solution schemes: staggered (de-coupled) and fully-coupled. A third approach mixes these two schemes by coupling some subset of the variables within a staggering scheme.

In staggered schemes, each of the PDEs is solved separately while holding the values of other variables constant. In this manner, a series of smaller linear problems are solved and the results of previous steps substituted into the current step. These methods are computationally attractive since the smaller linear systems require less work to solve; however, by neglecting some of the coupling terms between the equations, nonlinear convergence may be adversely affected. To overcome such difficulties, the problem can be solved in a fully coupled fashion, accounting for all relevant couplings. This approach can be computationally more expensive but is generally more stable.

Newton's method is often employed to solve the nonlinear systems arising from PDE approximations. Newton's method solves the nonlinear system via an iterative-refinement approach. A linearized form of the system is solved to obtain a refinement to the nonlinear approximation at each iterative step. For example, the drift-diffusion equations (2.1, 2.2, and 2.3) can be rewritten in the following general manner:

$$G_\Psi(\Psi, n, p) = 0 \tag{2.18}$$

$$G_n(\Psi, n, p) = 0 \tag{2.19}$$

$$G_p(\Psi, n, p) = 0 \tag{2.20}$$

With an appropriate initial guess for (Ψ, n, p), a new update ($\Delta\Psi$, Δn, Δp) is computed by solving the linear system:

$$\begin{bmatrix} \frac{\partial G_\Psi}{\partial \Psi} & \frac{\partial G_\Psi}{\partial n} & \frac{\partial G_\Psi}{\partial p} \\ \frac{\partial G_n}{\partial \Psi} & \frac{\partial G_n}{\partial n} & \frac{\partial G_n}{\partial p} \\ \frac{\partial G_p}{\partial \Psi} & \frac{\partial G_p}{\partial n} & \frac{\partial G_p}{\partial p} \end{bmatrix} \begin{bmatrix} \Delta\Psi \\ \Delta n \\ \Delta p \end{bmatrix} = \begin{bmatrix} G_\Psi \\ G_n \\ G_p \end{bmatrix} \tag{2.21}$$

The updates are used to refine the current solution approximation and the process is repeated until convergence is achieved. Many methods can be used to determine convergence. For instance, the process may be terminated once the magnitude of the largest update falls below a specified threshold.

To solve the linearized system at each step in the nonlinear process, a final piece of mathematical machinery is needed: a linear solution method. The choice of linear solution method is far from obvious and tends to be both application and problem-domain specific.

Regardless of the linear method chosen, exploitation of the sparse structure of the matrix is crucial for making the solution computationally tractable. The most stable and reliable approach is to solve the matrix directly using some form of Gaussian elimination [10]. Unfortunately, since the original sparsity pattern of the matrix must be augmented with non-zero fill-in during the solution process, direct methods tend to require a large amount of memory. To solve problems for which memory is the limiting resource or to solve well-conditioned matrices quickly, iterative methods can be used [22]. Iterative methods tend to preserve the original sparsity structure and require less memory than direct methods. However, they are less robust and may fail to arrive at a solution. Given the potential savings to be gained from iterative methods, much work has been done to develop preconditioning methods to improve convergence behavior. Unfortunately, truly robust methods have proved elusive for many applications, necessitating the continued use of direct methods for some problems.

Linear solutions are a computationally expensive aspect of PDE solutions. To reduce the linear solution cost, complexity may be added to the nonlinear solver. For instance, quasi-Newton updating schemes such as the Newton-Richardson [40] method may be implemented. By intelligently reusing previous factorizations to compute a series of nonlinear updates, a great amount of computation may be avoided, thereby speeding overall execution.

Linear solution methods are often independent of the nonlinear scheme employing it. To take advantage of the standard and well understood nature of linear algorithms, many software designers use existing library packages such as SPARSPAK [20] rather than implementing linear methods by hand. This allows great flexibility in the choice of methods as well as allowing the latest linear solver developments to be quickly integrated into PDE applications.

2.2.3 PDE Approximation (Matrix Formation and Assembly)

The physical model evaluation and matrix formation are the program components which apply the PDE approximation scheme in the computer model. Generally, to make this phase of the computation straightforward and regular, each component of the discretized problem is evaluated separately. For example, the FE method uses element-by-element evaluation techniques. Although the computation may be done within a single element loop, the domain is typically subdivided into blocks of elements (Figure 2.7). Elements may be divided into groups with like material type or similar geometry to improve code organization or ease model implementation. Likewise, the elements may be divided such that each block consists of elements with non-overlapping data accesses in order to improve code performance on various computer architectures. Figure 2.8 shows the high-level code structure of the element evaluation. This two-level loop structure is characteristic of most PDE solvers. (In the case of FD or FV, processing takes place on the nodes or volumes, rather than the elements, but the code structure is analogous.)

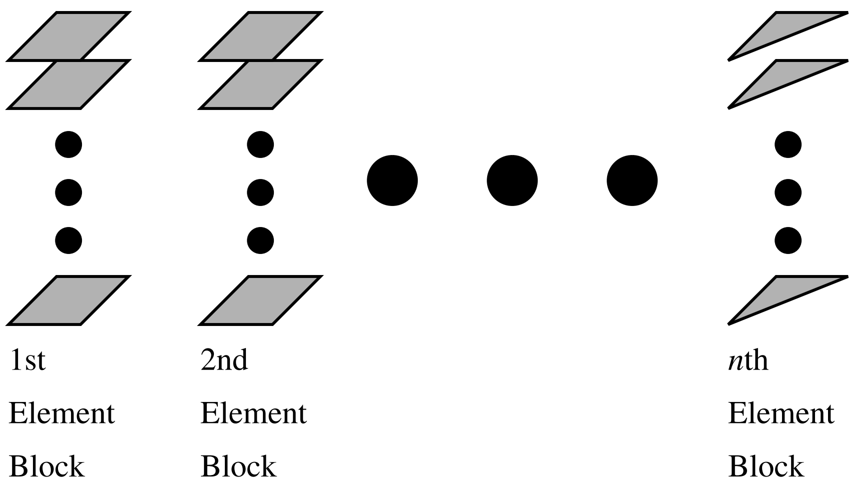

Figure 2.7 Blocked element organization used in model evaluation and matrix formation.

```
for block = 1 to number_of_element_blocks
    for i = 1 to block_size(block)
        evaluate_and_assemble(element(i))
    endfor
endfor
```

Figure 2.8 Pseudo-code for element assembly.

It is in these modules that physical properties are taken into account. For example, the approximation of Poisson's equation is dependent upon the charge in the domain. Charge, in turn, depends upon the materials comprising the domain, boundary conditions (externally supplied charge), as well as the physical phenomena governing charge motion and charge creation (electron and hole generation/recombination mechanisms). Relevant physical models to account for these effects must be present in the approximation modules.

In general the evaluation of physical models is purely local (within the element); however, some physical phenomena may span an area larger than a single element and correct evaluation may require access to non-local data. One case is shown in Figure 2.9. Assume element **E** maintains a model which depends upon some property p and data concerning p is distributed among neighboring elements. To be evaluated correctly, the model in element **E** requires access to the neighboring elements' information concerning property p. An example arising in many applications occurs at material boundaries. Material junctions often create anomalous behavior and elements must be aware of the materiel types of surrounding elements to account for these anomalies.

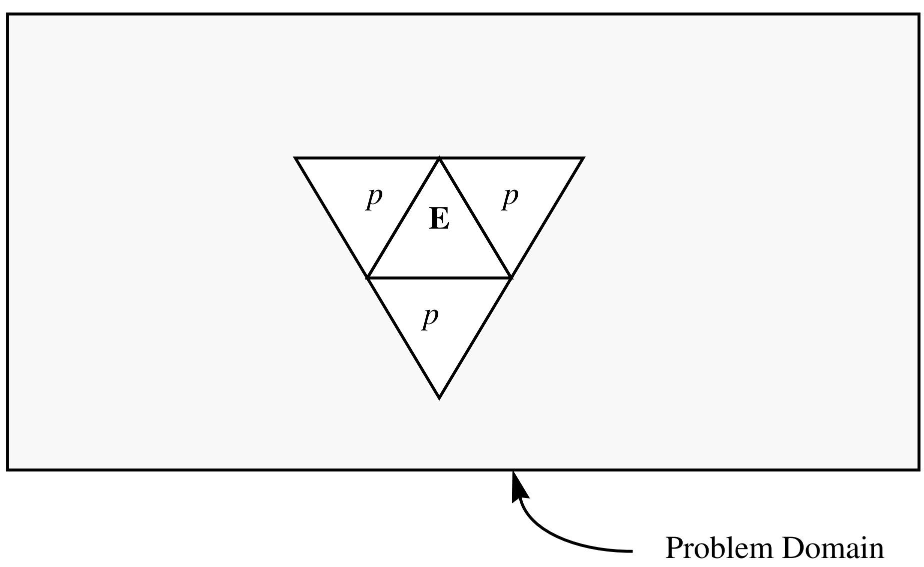

Figure 2.9 Data non-locality in model evaluation. In this case, triangle **E**'s assembly is relies on property p of neighboring triangles.

In addition to applying the requisite physical models, the matrix formation and physical modeling modules also evaluate and apply the boundary conditions governing the PDEs. The proper choice and correct use of boundary conditions varies considerably with approximation method and a full discussion is beyond the scope of this dissertation. However, it is useful to discuss examples of the data non-localities generated by boundary conditions. Figure 2.10 depicts the external connectivity imposed by a boundary condition. This extra connectivity often creates extra constraints upon the system. To account for these constraints, each element regulated by this condition may require data residing in the other elements attached to the boundary condition.

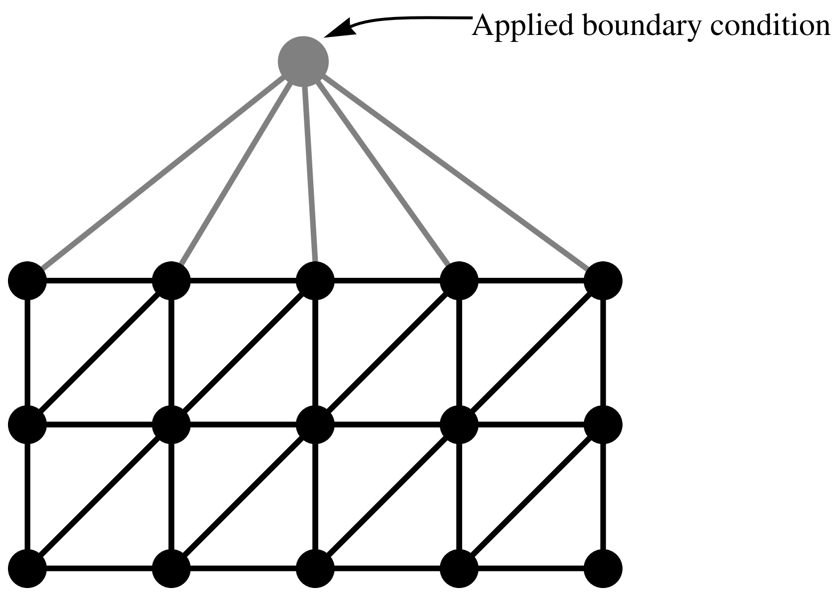

Figure 2.10 Data non-locality caused by boundary conditions.

2.2.4 Data Flow between Major Computational Modules

To accomplish its task, each program module may employ an appropriate subset of the algorithms introduced in the preceding sections. Many trade-offs involving memory consumption, computational demands, algorithmic stability, and solution accuracy must be considered during the design and development of a PDE solver. A full-scale application to solve PDEs links appropriately chosen functional modules together in a meaningful fashion. Fortunately, each of the major computational modules is largely self-contained. More importantly, the data items passed between the modules are conceptually well defined, and the modules usually interact with one another through a set of rigid interfaces. Figure 2.11 shows the data flow between the major computational components in a PDE application. (The UI has been omitted from the figure since it is not normally active during the solution process.) The nonlinear solver drives the solution process and iterates until the nonlinear convergence criteria have been met. During computation, the nonlinear solver will request a new matrix and residual from the discretization module. In turn, these objects are passed to the linear solver to compute the next update to the approximate nonlinear solution. All of these modules may require access to the global, user-defined solution parameters managed by the UI. The modular aspect inherent in most PDE solvers allows individual modules to be updated or entirely replaced without affecting other parts of the application.

In general, all of the computational modules shown in Figure 2.11 contribute significantly to the execution time of the overall application. In some cases, linear solution requires the majority of the solution time. However, for codes with elaborate material models, sophisticated approximations, or complex nonlinear solution schemes this is untrue. To illustrate, the relative execution times of the linear solver and the nonlinear solver and matrix formation steps for a typical PDE solver are shown in Table 2.1. The problems were run on a state-of-the-art high-performance workstation (IBM RS6000/ Model 590). The grid sizes vary from roughly 6,000 nodes to more than 500,000 nodes. The linear solver is known to scale more poorly with increasing grid size than the other modules and takes a larger component of execution time as the grids become larger. However, considering both average and extreme grid sizes, the linear solver is not the only computationally intensive task.

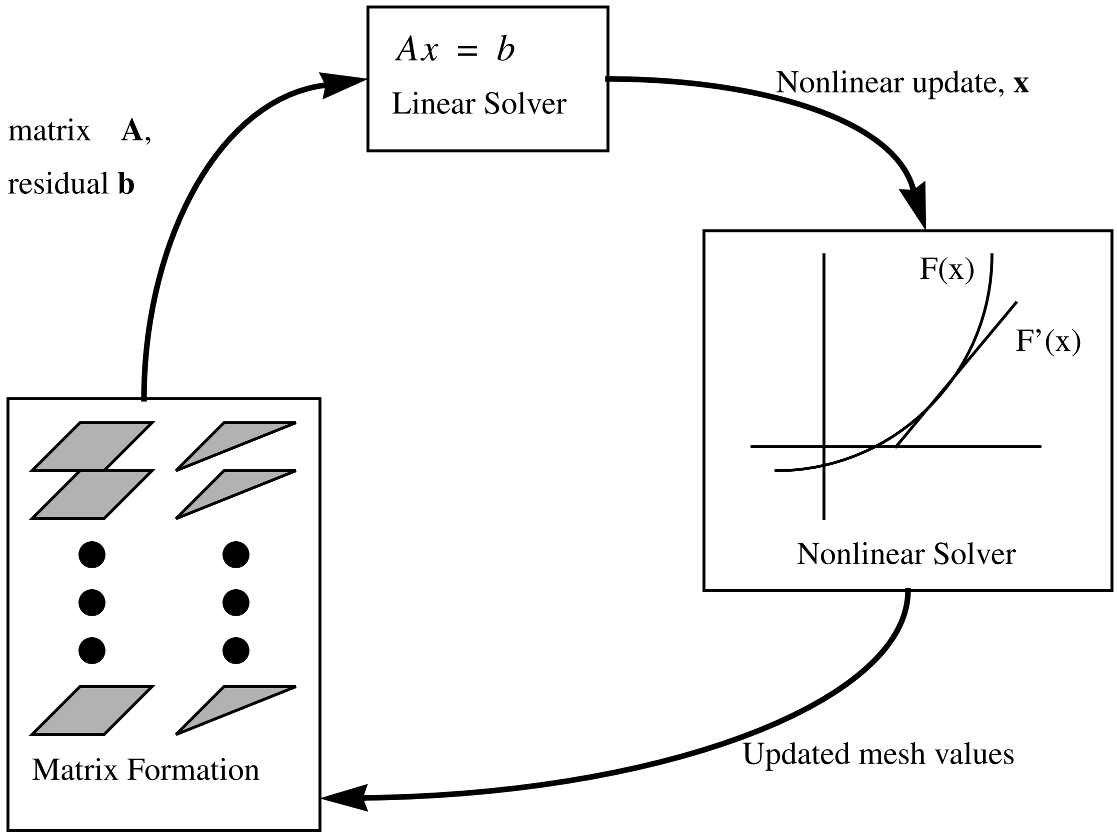

Figure 2.11 Data flow between major components in a PDE solver. The process shown is repeated until convergence is achieved in the nonlinear solver.

As parallel computers have become available, one common approach to improve the performance of existing PDE solvers is to replace an existing linear solver with a parallel linear solver. However, this approach may be of limited utility. Neglecting the issue of data distribution and assuming that a perfect parallel linear solver were available, applying Amdahl's Law [30] shows the limited maximum speedups for these realistic grid sizes. Clearly, restructuring all of computational modules to run in parallel is vital in order to exploit the potential of parallel computers.

Table 2.1 Relative execution times of the linear solver and the nonlinear solution and discretization.

	Linear Solution	Nonlinear Solution & Discretization	Maximum Speedup w/Parallel Linear Solver
CMOS Inverter (2,000 grid points)	29%	71%	1.41
Bipolar Transistor (6,000 grid points)	46%	54%	1.85
CMOS Inverter (11,000 grid points)	42%	58%	1.72
Photo-Emitter (512,000 grid points)	67%	33%	3.03

2.3 Coarse-Grain Decomposition for PDE Solution

As stated in Chapter 1, PDE solutions for many application areas demand computational resources with the highest available performance; therefore, the use of parallel computers to improve simulation capabilities has become increasingly important. Furthermore, as Table 2.1 has shown, parallelization of an entire PDE application must be accomplished in order to provide a significant performance increase. Accomplishing this goal not only requires appropriate data decomposition but also necessitates efficient program decomposition. Data and control inconsistencies resulting from these decompositions must carefully and conscientiously be resolved to insure globally correct solutions. An algorithm-driven, coarse-grain approach to decomposition results in a flexible and intuitive distribution of data and code. This high-level decomposition of both the code and the data into a small-to-moderate number of sub-problems is appealing for two reasons. First, a high-level decomposition will allow the reuse of existing data models, data structures, algorithms, and code structures without significant changes. (This eases the parallelization of existing codes as well as the development of new ones.) Secondly, dividing the problem into a small number of coarse-grain components allows a straightforward mapping onto the most popular and widely used parallel architectures. (These parallel computers are built with a small to moderate number processing elements, each suitable for tackling a coarse-grain subdomain.)

2.3.1 Data Decomposition

A PDE discretization is effectively a fine-grain decomposition of a problem domain into a set of small, easily soluble pieces. In typical applications, a slightly coarser block structure is created to group pieces into larger, more efficient computational units. To accommodate parallelism, this approach may be extended to create an even higher-level set of *parallelizable* computational units. These blocks, the parallel subdomains, are disjoint pieces of the global simulation domain. Using well-known grid decomposition techniques [13], the simulation grid can be broken into disjoint pieces to create the parallel subdomains. (For example, Figure 2.12 shows a 16-way grid decomposition of the grid introduced in Figure 2.1.) Data dependencies between the parallel subdomains will exist across the shared boundaries. Data inconsistencies arise when information once easily accessible through the consistent, single memory view of the serial application is no longer directly available due to data distribution. If these data dependencies can be satisfied through inter-domain communication, a global solution identical to the serial solution can be computed.

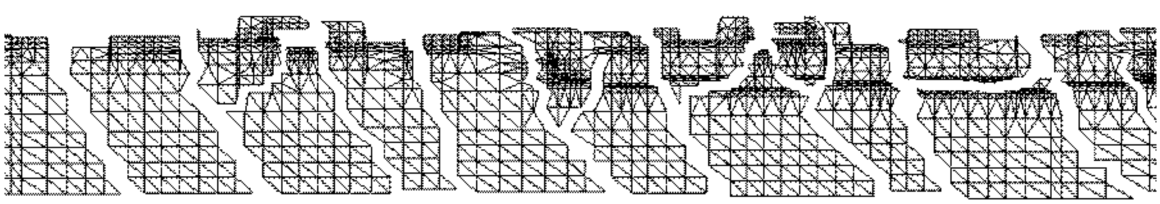

Figure 2.12 16-way coarse-grain grid decomposition of a CMOS inverter.

2.3.2 Program Decomposition

In the coarse-grain model, each processor of the parallel machine runs a copy of the application to solve one subdomain of the partitioned grid. Conceptually, this approach attempts to parallelize computation at the newly created subdomain level (i.e., on the outermost loop rather than, as many automatic tools, at the innermost loop). For example, Figure 2.13 shows the outer subdomain loop implicit in the distributed matrix formation module.

```
doacross(subdomains)
    for block = 1 to number_of_element_blocks
        for i = 1 to block_size(block)
            evaluate_and_assemble(element(i))
        endfor
    endfor
enddo
```

Figure 2.13 Pseudo-code for distributed element assembly.

The wide range of algorithms employable at each phase of PDE solution (Section 2.2) forces parallelization to be application-domain-dependant; nevertheless, a general high-level structure is evident. The decomposed application structure is shown in Figure 2.14. Each of the three components in the structure serves a unique function and arises naturally from the existing nature of PDE applications. In this diagram, vertical lines represent the flow of control between the major components. Control is effected through the transfer of data symbolizing either a computational request or completion of a computation. On the other hand, horizontal lines represent the flow of data needed to resolve data and timing inconsistencies created by the partitioning.

At the highest level in the application (above the nonlinear solver), a global controller is needed to maintain a global, unified problem view and to provide a consistent interface to the user. Since the global controller effects control and distributes the problem from the highest possible application level, maximum parallelism can be exposed. To simplify code structure and accommodate parallel architectures without direct user access to the parallel nodes, the global controller may be developed as a separate program which communicates with the parallel solution modules. Since the UI routines take negligible time and are inherently serial, no measurable performance effect is expected in the parallel code; therefore, the traditional PDE UI module provides a good starting point for the creation of a global controller - with the addition of functionality to control and communicate with the computational modules. To strictly choreograph the solution process, the global controller must provide a global "clock" to effect synchronization and insure that the subdomain solvers act together. Finally, some method for breaking apart simulation grids and managing the pieces must be added.

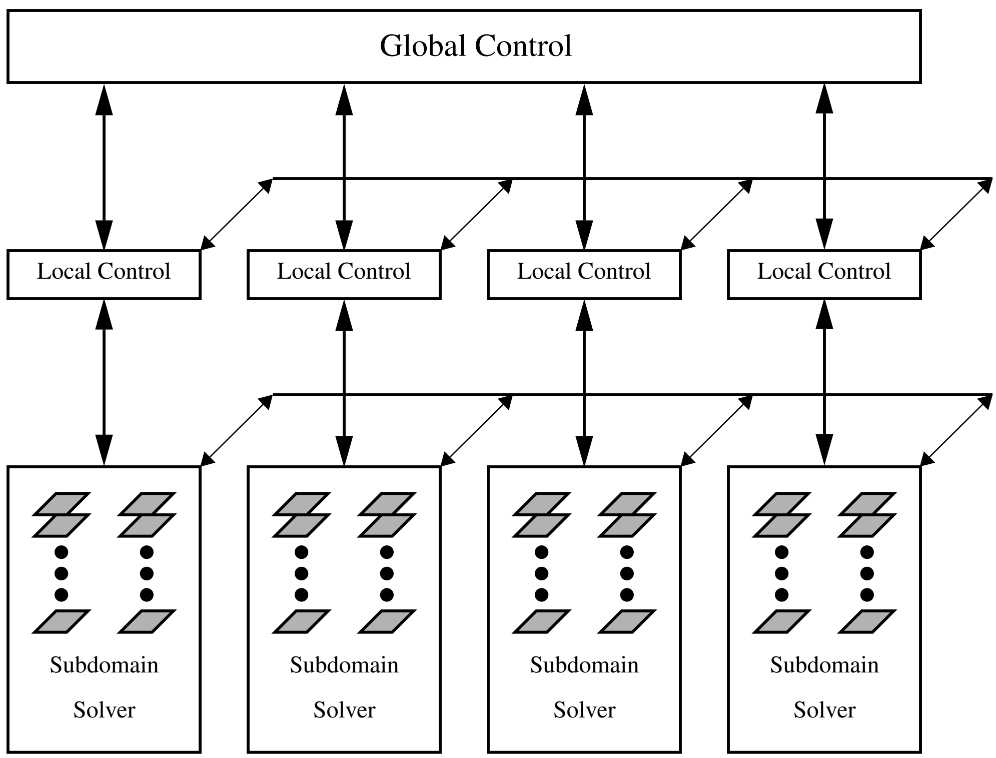

Figure 2.14 Decomposed PDE-program organization.

A local controller on each processor manages the finer details of the solution process and insures coherence and synchronization within the nonlinear solution process. The distributed nature of computation may require synchronization at each nonlinear step as well as the exchange of data such as boundary condition information, nonlinear residuals, and convergence norms.

At the lowest level in the application model, the subdomain solver is responsible for performing the requisite mathematical steps to solve the distributed domain. To accomplish this, each processor runs traditional serial-style modules modified to resolve non-local data dependencies. The functionality of the subdomain solver includes matrix formation, physical model evaluation, linear solution, and the computational portions of the nonlinear solution. Non-local data may be required within any of the computational modules and includes boundary condition data, nonlinear residuals, non-local material model data, and matrix updates in the linear solution. In general, non-local data may be divided into two types: time-invariant and time-variant. Likewise, two methods exist for resolving non-local references: replication and communication. Data replication reduces communication, but increases memory size and is mainly suitable for reducing redundant communication for time-invariant data. Data communication is used for time-variant data and for time-invariant data when memory is limited. Table 2.2 summarizes the preferable approach to resolve the various types of data non-localities. Properly classifying non-local data and efficiently resolving references can have a large performance and memory impact on the parallel application. Figure 2.15 shows one possible high-level code structure for element evaluation which includes the communication step to resolve references. It is assumed that replicated data may simply be accessed from a local buffer within the element loops and causes no major changes to the code.

Table 2.2 Guidelines for resolving non-local data dependencies.

	Time-variant	Time-invariant
Small data transfer	Communicate	Replicate
Large data transfer	Communicate	Communicate

```
exchange_time_variants()
doacross(subdomains)
    for block = 1 to number_of_element_blocks
        for i = 1 to block_size(block)
            evaluate_and_assemble(element(i))
        endfor
    endfor
enddo
```

Figure 2.15 Pseudo-code for distributed element assembly.

2.3.3 Applicability to Existing Codes

Often, it is desirable to use existing codes as building blocks for parallel codes since most users are unwilling to discard the knowledge and expertise captured by their existing “dusty-deck” serial PDE solvers in exchange for a faster yet unproved and unfamiliar parallel code. The coarse-grain approach to parallelization preserves knowledge inherent in “dusty-decks” and pleases users by maintaining a familiar environment while providing an improved solution capability. During a typical code’s lifetime, great care has been taken to validate the code as well as to improve and calibrate the physical models. In fact, industrial users have often invested considerable calibration effort to further improve accuracy. Moreover, long program lifetime allows a large, experienced, and satisfied user base to be established. These users understand the subtleties of the simulator and have developed unique and sophisticated solution strategies. Finally, both the long lifetime and large user base mean that many bugs and inconsistencies in the code have been exposed and either corrected or worked around. Developing a new application would force users and developers to repeat an enormous amount of non-trivial and time-consuming work beyond initial code development. These facts make strong arguments for parallelizing existing codes provided that the effort is not too great and the parallel code maintains consistency with the original serial code. This requires that changes to the existing code be minimized and localized. Neither the data structures nor the algorithms can be fundamentally altered if the value of existing codes is to be preserved. A coarse-grain approach to parallelization allows existing codes to be encapsulated into the subdomain solver without significant modifications. The code can then be strictly choreographed from above by the local and global controllers. Communication routines must be added to resolve non-local references within the encapsulated (and now distributed) code.

2.3.4 Suitable Programming Model

Many standards exist for the development of serial codes, allowing straightforward porting of code across various hardware platforms. Unfortunately, the emerging parallel hardware market has yet to establish comparable standards. Nonetheless, the coarse-grain parallelization approach provides a flexible conceptual framework which, with the proper choice of programming model, can be made portable with minimal effort. Moreover, to insure that the parallelization of existing codes can be done quickly, smoothly, and without major changes to the program, it is crucial to select a parallel programming model amenable to existing PDE codes.

The dominant parallel programming paradigms are data-parallel [27], shared-memory [63] and explicit message-passing [62]. Data-parallel programming uses new parallel languages like HPF [55] and requires the parallelism to be made explicit in the data structures and code constructs. Thus, modifying the data and code structures to be compliant with this model requires significant recoding of existing applications. More troubling is the lack of standards in data-parallel programming potentially forcing significant recoding to port across hardware platforms. These concerns preclude a data-parallel implementation of the coarse-grain approach.

Shared-memory programming allows the use of standard programming languages and thus the reuse of existing codes. Unfortunately, like data-parallel programming, shared-memory programming offers few standards. Also, the shared-memory programming model is not available on the many distributed-memory message-passing computers. Again, lack of portability makes shared memory programming an unrealistic option.

Explicit message-passing programming allows both the use of existing programming languages and provides reasonable standards. More importantly, this programming model is available on most commercial hardware platforms, making porting straightforward. In fact, the recent adoption of the Message Passing Interface (MPI) standard [24] by hardware vendors insures the availability of the message-passing paradigm on most architectures. Accordingly, explicit-message-passing programming is the most attractive choice for the implementation of coarse-grain PDE applications.

Remark: As discussed earlier, linear solution modules from standard libraries are often used in PDE applications. Unfortunately, the close coupling between linear solver and application implicit in serial PDE solvers not longer exists in a distributed computing environment. Thus, parallelizing these libraries can be problematic in many cases. If appropriate parallel libraries exist at all, using them is often difficult. In fact, incompatibilities with matrix representation and data distribution often preclude the use of parallel libraries. Also, many parallel linear solvers are architecture-specific and are not portable. The issues surrounding parallel linear solvers in PDE applications will be discussed in Chapter 4.

2.4 High-Performance Parallel Architectures

Most commercially available parallel computers utilize a small to moderate number of high-performance processors and support a message-passing library. The coarse-grain software approach already described is well matched to this hardware structure. The main conceptual difference between most commercially relevant parallel machines lies in the absence or presence of a globally addressable memory space. This difference impacts the available programming models. Machines without global shared memory require the use of a message-passing primitives for communication and will be referred to as “message-passing computers.” Machines with global shared memory generally communicate via the shared memory space and will be referred to as “shared-memory computers.” As described above, message-passing libraries such as MPI allow message-passing codes to run directly on shared-memory computers. Also, the message-passing model can be emulated on shared-memory machines through the coordinated use of shared communication buffers.

2.4.1 Message-Passing Parallel Computers

Message-passing parallel computers consist of a collection of processors, each with a local memory, connected via a communication network as shown in Figure 2.16. This definition is broad enough to include collections (clusters) of workstations connected a network (e.g., Ethernet) as well as the dedicated parallel processors connected with high-performance switches discussed in this section. Due to the strict compartmentalization of resources, processor-to-processor communication must be handled through explicit send/receive and global synchronization mechanisms. This functionality is provided through some form of message-passing library. The three parallel architectures used in this dissertation (the Intel iPSC/860, the Thinking Machines' CM-5, and the IBM SP1/2) are all message-passing machines.

The Intel iPSC/860 (also referred to as the iPSC) system, introduced in 1990, is a collection of processing elements (PEs) connected by a hypercube network [28]. In an n-dimensional hypercube, each PE has n communication channels. In this topology, one PE is never more than n hops away from any of the other 2^n-1 processors. (The number of processors must be a power of two.) Each PE of the iPSC is an Intel i860 microprocessor with up to 32 megabytes of local memory. Each PE runs a minimal operating system, the Node Executive (NX) [62]. The lack of virtual memory support in NX limits the size of problems which can be solved. NX provides message passing functions to allow PEs to communicate with one another. Message latency is roughly 200 microseconds for messages longer than 100 bytes. Communication bandwidth is about 2 Mb/s (bi-directional). As with many early parallel systems, the PEs cannot access the "outside world" directly. Consequently, the processing nodes are attached to and controlled by a serial host. The serial host is an Intel 80386 workstation running the Unix operating system.

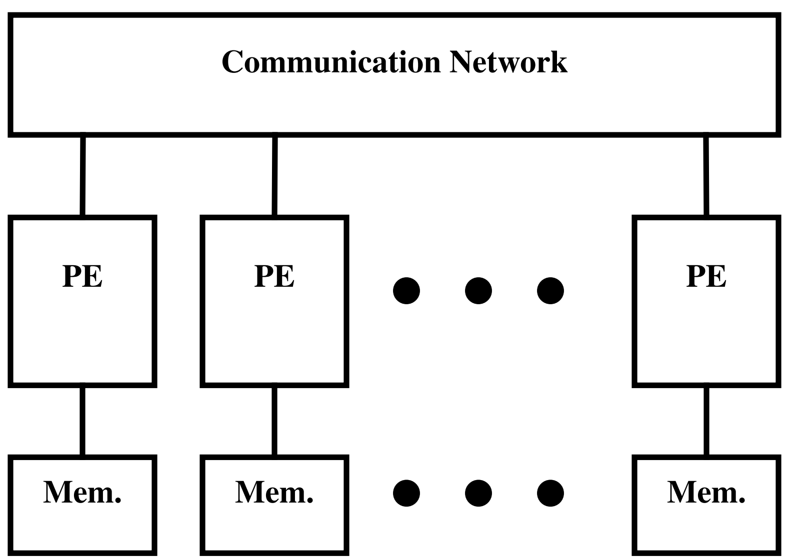

Figure 2.16 Abstract sketch of a distributed-memory parallel computer.

The Connection Machine CM-5 by Thinking Machines Corporation [57] was first introduced in 1991. Each processing node consists of a SUN SPARC processor which controls four attached vector units with 32MB of memory and a network interface. A CM-5 system also contains one or more front-end control processors. The CM-5E, introduced in 1993, uses higher performance SUN Viking processors instead of the original SPARC processors. The CM-5E also has an increased memory of 128MB per node. The front-end control processors and processing nodes are interconnected by a communication network with a 4-ary fat-tree structure [33] providing 18MB/second communication bandwidth with 82 microsecond latency. The control processors run an extended version of UNIX; each processing node runs an operating system microkernel. Although the CM-5 was originally designed to support the data-parallel programming model, it also supports the message-passing programming model through its CMMD message passing library [56]. Unfortunately, the vector units can only be utilized by programs written using a data-parallel language such as Connection Machine FORTRAN [55]. This means that message-passing programs are confined to the SPARC (or Viking) processor and unable the access the high-performance vector units. Although this handicaps message-passing programs, the machine still offers reasonable performance.

The IBM PowerParallel SP1 and SP2 systems, introduced in 1993 and 1994 respectively, offer a collection of high-performance RS/6000 workstations connected via a local Ethernet and/or a dedicated high-speed switch. The high-speed switch is a multi-stage crossbar [60]. On the SP1 series, the switch provides communication channels with 80 microsecond latency and 64 MB/second bandwidth; latency is 40 microseconds and bandwidth is 32 MB/second on the SP2 switch. Each PE runs the full AIX operating system and is typically configured with large amounts of local memory as well as local swap and local scratch disk space. While peripheral nodes may be added to the PowerParallel system as disk servers, compile servers, etc., each PE is directly accessible directly to the user (without using a front-end control processor) and can be used as a serial workstation. The SP1/2 series supports a number of message passing libraries [59], including PVMe, MPL, and MPI.

Table 2.3 lists the message-passing functions required by the coarse-grain model on each of the three message-passing computers described above. By creating coarse-grain parallel PDE solvers with this minimal set of primitives, straightforward portability is assured. Often, porting is accomplished by simply translating the original messaging calls into the analogous calls on the target architecture. Noting the similarity of most message-passing libraries and recognizing the need for standardization, most hardware vendors also provide the new MPI standard as well as their proprietary libraries. (The necessary MPI library calls are very similar to MPL calls.)

Table 2.3 Equivalent message directives for message-passing libraries.

	NX	CMMD	MPL
Blocking Send	csend	CMMD_send_block	mpc_bsend
Non-blocking Send	csend	CMMD_send_noblock	mpc_send
Blocking Receive	crecv	CMMD_receive_block	mpc_recv
Reduce/Combine	*gxfunc*	CMMD_combine	mpc_combine
Synchronization	gsync	CMMD_sync_nodes	mpc_sync
Processor ID	mynode	CMMD_self_address	mpc_environ
Machine Size	numnodes	CMMD_partition_size	mpc_environ

2.4.2 Shared-Memory Parallel Computers

Parallel computers having a shared address space may actually have physical memory distributed along with the PEs as in Figure 2.16. In this case, hardware support is provided to access remote memory. Examples of this architecture include the Stanford DASH [34] and the Cray T3D [58]. Another popular architecture is shown in Figure 2.17 and is known as a symmetric multiprocessor (SMP). In an SMP, the shared memory is not distributed with the processors, but resides in a global pool, providing uniform access. Memory access is often through a shared bus. The bandwidth of the shared bus often limits scalability to moderate numbers of processors. Nevertheless, SMPs are popular not only as parallel processors but also as throughput computing resources shared by large workgroups. Thus, SMPs will likely be a major component of parallel computing resources for the foreseeable future.

One popular SMP is the PowerChallenge [63] from Silicon Graphics Corporation. Each processing element is a MIPS R8000 microprocessor. Up to 18 processors can be support in the current PowerChallenge architecture. The processors are connected to the main memory (up to 16 GB) by a high-speed bus. This bus provides up to 1.2 GB/seconds sustained transfer rate. Hardware support maintains cache coherency among the processors [2]. Both shared memory and message passing programming paradigms are supported. Both the PVM [4] and MPI message passing libraries run on the PowerChallenge and use efficient shared memory copies to transfer messages. Thus, most message-passing programs can be easily adapted to run on the PowerChallenge.

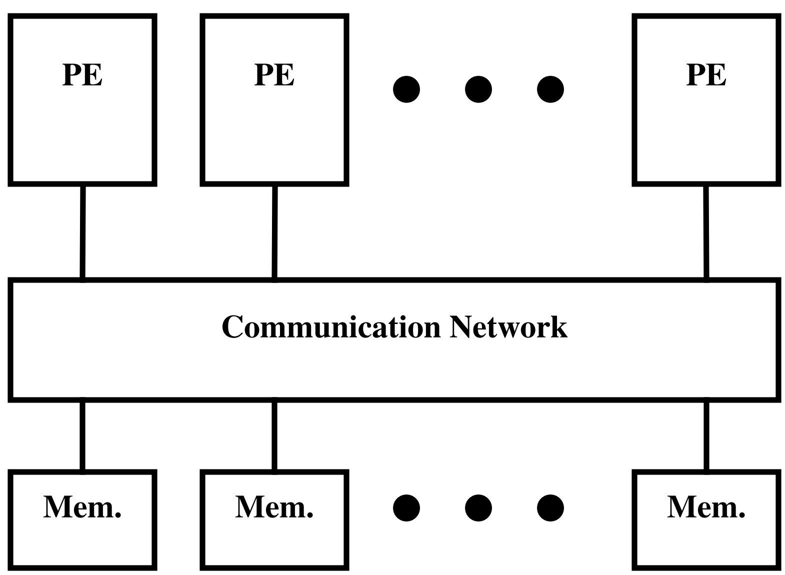

Figure 2.17 Abstract sketch of an SMP parallel computer.