Chapter 3

Coarse-Grain Parallelization of Semiconductor Applications

In this chapter, the creation of parallel PDE applications using the coarse-grain model is described. The coarse-grain methodology has been applied to two very different semiconductor device simulators. One code is an example of a “dusty-deck” PDE solver using fully implicit direct solution techniques. The second application is typical of modern finite element software and utilizes implicit iterative solution methods. After a brief introduction to the first application, its parallelization is presented in-depth and the practical issues involved are analyzed. Once the parallel framework created for data and control has been described, its reuse for the parallelization of a second code is briefly discussed. In both cases, application portability is shown and parallel results are presented for several architectures.

3.1 Device Characterization: The PISCES Simulator

PISCES is a large and complex application which solves problems strategic to the semiconductor industry. The code encompasses many areas of research in both physics and computational mathematics. It is a two-dimensional, two-carrier semiconductor device modeling program developed primarily at Stanford University over the past fifteen years and is well known throughout the semiconductor industry. The program is available from Stanford or from one of several Technology CAD vendors who support the code commercially. The program structure of the PISCES simulator adheres to the general PDE program model discussed in Chapter 2.

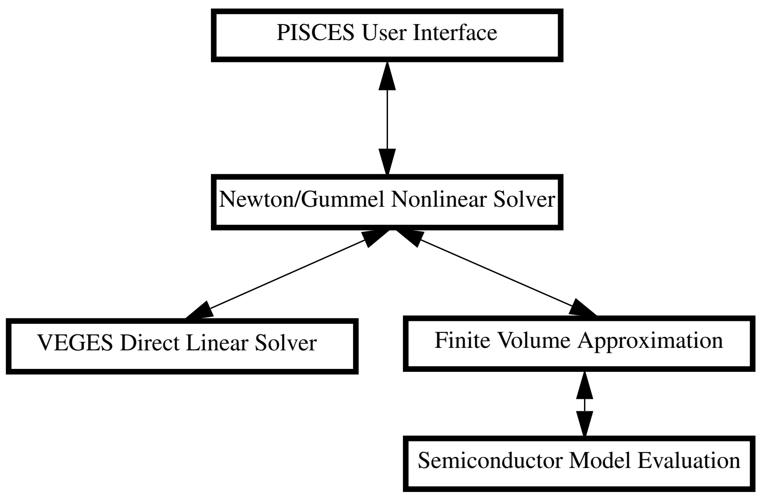

Figure 3.1 PISCES program structure.

PISCES solves Poisson's equation and the semiconductor continuity equations using the drift-diffusion formulation presented in Section 2.1. These coupled nonlinear PDEs are approximated using a finite volume formulation on a non-uniform triangular grid as discussed in Section 2.1.1. (See Price [43] for a detailed discussion of the FV method as applied to the drift-diffusion equations for semiconductors.) The coarse-grain structure of the simulator is shown in Figure 3.1. The user interface is built around a simple command-line parser which employs a dispatch loop to control the solution process. The UI also performs file I/O functions and displays simulation results via plotting utilities. The nonlinear solver supports a wide range of solution techniques and contains both a fully-coupled Newton [21] solver and staggered Gummel [25] schemes. Within the FV approximation, PISCES offers a large and diverse set of physical models. Improvements and additions to the physical models are the most active areas in the continuing development of the simulator. During the nonlinear solution process, the sparse linear system is assembled in a triangle-by-triangle (finite element) fashion using a subset of the available physical models. It has been shown by Rafferty [45] that the matrices arising in this type of device simulation are extremely ill-conditioned and not easily solved using iterative techniques. Therefore, the sparse system of linear equations is solved using the VEGES solver [5], an optimized, public-domain sparse direct solver.

The CMOS inverter grid shown earlier (Figure 2.1) is representative of PISCES simulation grids. This coarse grid contains roughly 1,600 nodes and 3,000 triangles. The structure was created using standard one micron layout rules and was derived from the model of an SRAM cell. A simple simulation to compute the DC terminal characteristics was run using a finer mesh than shown (roughly 8,000 nodes and 15,000 triangles) for improved accuracy. After reading the grid and setting the proper parameters, the simulation brings the power supply to 5V and then ramps the input gates from 0V to 5V. The output is shown in Figure 3.2. As expected in an inverter circuit, the output experiences a rapid transition from 5V to 0V as the input voltage increases.

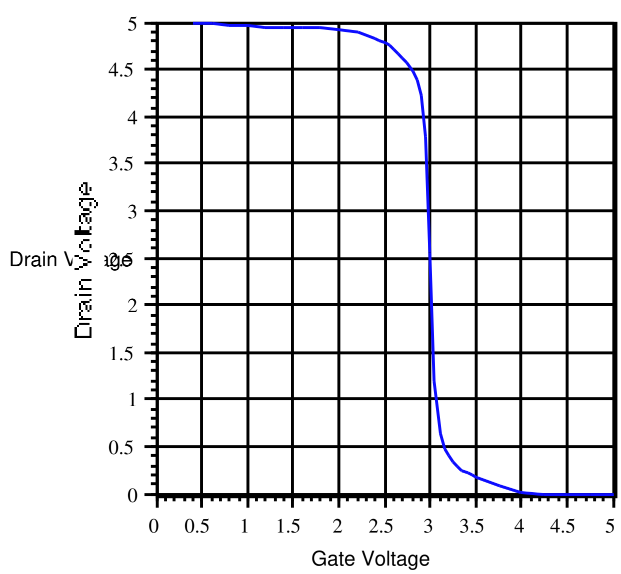

Figure 3.2 CMOS inverter DC characteristic.

This simple simulation required roughly twenty minutes on a state-of-the-art high-performance workstation (IBM RS/6000 Model 590). Using a finer mesh containing nearly 19,000 nodes and 36,000 triangles, the simulation required roughly one and one-half hours. Simulations to perform more detailed analyses of this structure can easily take several days to complete. For example, if multiple simulations were performed as part of a manufacturability study, several weeks might be needed. A significant boost in the performance of PISCES will be needed if it is to keep pace with current and future demands. As discussed in Chapter 1, the use of scalable, high-performance parallel architectures to deliver that performance will become strategically important in the continued success of large-scale device simulation.

3.2 PISCES Parallelization

It was observed by Lucas [18] that for small simulation grids (less than 2000 grid points) PISCES normally spends more than 95% of its runtime solving the coupled nonlinear device equations. The remaining fraction of time is spent in the UI. Within the nonlinear solver roughly half the time is spent computing linear solutions to obtain updates to the nonlinear system. The other half of the nonlinear solution time is spent creating initial guesses, forming and assembling the linearized matrices, evaluating physical models, and applying nonlinear updates. More recent analysis shows nonlinear solution times grow to be more than 99% of the total runtime for larger grid sizes. As grid sizes increase, the linear solver scales more poorly than other parts of the code; nevertheless, for structures containing hundreds of thousands of grid points, 30-35% of the runtime remains outside of the linear solver. (Table 2.1 in the previous chapter summarizes the relative execution times of major PISCES routines and demonstrates the necessity of parallelizing all phases of the nonlinear solution.)

Before blindly initiating the development of the parallel application, the expected effort should be estimated and weighed against the expected benefits. As discussed in Chapter 1, parallel grid-based PDE solvers have had measurable success compared to traditional high-performance serial workstations and can be expected to achieve an order-of-magnitude reduction in simulation runtimes using sixteen processors. On the other hand, the wisdom of reusing old, poorly structured code rather than creating a new application must be questioned. During its fifteen-year life-span several generations of researchers have modified the code. The current version of PISCES consists of approximately 40,000 lines of FORTRAN-77. PISCES has become unwieldy due to this continual development. Despite its inelegant program structure, great care has been taken to validate the code as well as to improve and calibrate the physical models. In fact, industrial users have invested considerable effort calibrating PISCES with experimental data to further improve accuracy. Moreover, its long lifetime has allowed a large, experienced, and satisfied user base to be established. These users understand the subtleties of the simulator and have developed sophisticated solution strategies unique to PISCES. 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 the current code 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. Fortunately, the coarse-grain parallelization methodology presented in Chapter 2 allows the exploitation of parallelism through the encapsulation of these "dusty" modules, preserving the value of the code.

3.2.1 Previous Parallelization Efforts

In earlier work, Lucas created Parallel PISCES to showcase his parallel linear solver developments [35]. Rather than attempting a full-scale parallelization, "Parallel PISCES was designed to demonstrate that the throughput of PISCES could be improved by exploiting concurrency." The code was developed for an early hypercube architecture inadequate for large-scale application development, the Intel iPSC/1. (The iPSC/1 was comprised of Intel 80286 processors each with 0.5 Mbytes of memory and controlled by a front-end processor.) Although this demonstration was successful, many important issues were not addressed. Both the development goals and the available computing platform limited the general-purpose utility of the resulting parallel application. The most limiting aspect of this prototype code was the maintenance of a global problem image on each processor. (All data other than that used by the linear solver was replicated on every processor.) This replication of data allowed communication to be limited to the linear solution phase of the simulation. However, redundant work was performed on each processor to insure a consistent state. Hence, two problems were encountered. First, the wholesale replication of data wastes memory and severely limits the problem sizes which can be addressed. Secondly, the redundant work introduces a serial bottleneck which becomes more severe as the number of processors increases. Thus, this approach is not broadly scalable with problem size or with the number of processors. In addition to the scalability limitations, domain decomposition in Parallel PISCES employed a simple geometric bisection algorithm suitable only for regular rectangular grids. This further restricts the number of meaningful problems that can be solved. Finally, I/O and control functions (a simpler version of the coarse-grain global controller) were performed by the front-end processor of the iPSC/1. (The front-end processor communicated with the hypercube through a special host/node communication function unique to the iPSC.) The front-end processor of the iPSC/1 was notoriously underpowered (in terms of both memory and CPU) and inadequate to provide support for large problems. As not all parallel machines have dedicated front-end processors, the use of the iPSC/1 front-end to manage computation, while a reasonable first approach, again limits both the problems that may be addressed and the portability of the resulting application.

3.2.2 Applying the Coarse-Grain Parallel Model

Applying the coarse-grain parallelization approach described in Chapter 2 should remedy many of the shortcomings inherent in the Parallel PISCES prototype. By utilizing general domain decomposition methods and introducing communication rather than replication into many phases of the computation, the resulting coarse-grain parallel code (known as PISCES-MP) should provide scalable, portable, and general-purpose solution capabilities. The following sections discuss in detail the steps required to create PISCES-

MP by adapting PISCES to the coarse-grain parallel model. Initial development took place on the Intel iPSC/860 hypercube (described in Section 2.4.1) and the design choices dictated by this architecture are discussed. Portability of the resulting application to several parallel architectures is subsequently shown.

3.2.2.1 Global Control: Separating Serial Code from Parallel Code

Creation of the coarse-grain global controller does not require that the solution process run in parallel and may precede the actual solver parallelization, simplifying code management. Therefore, before embarking upon the parallelization of the device solver, the PISCES code was bifurcated into two serial programs: a serial UI program and a serial solver application (SA). This initial program decomposition resolves the data dependencies between the UI and serial SA, sets up a communication mechanism between the two, and creates a coarse-grain choreographer to coordinate their interaction. To address one shortcoming of the earlier work, the global controller (front-end program) should run on a computing platform with sufficient CPU and memory to allow solution of meaningful problems. Thus, the global controller was designed to be run on any remote workstation with network access to the parallel computer. The serial SA runs on one node of the parallel computer.

To insure generality, a transportable data connection between the global controller and the solver module is required. A communication package using TCP/IP [61] was created to facilitate the data transfer. TCP/IP provides great flexibility at the price of a very slow connection. (The data transfer rate is roughly 65kB/s in the development network environment.) However, the communication package can easily be replaced when faster communication resources are available. The main complication involved accommodating the differing machine-word byte orderings of the Intel hypercube and the various workstations. In the communication routines, the receiver is responsible for byte swapping data upon receipt, if necessary. To simplify this procedure, data was separated into the five types used by PISCES: integer, real, double precision, character, and logical. Similarly typed data are grouped into large blocks for transmission between the UI and SA. This makes the swapping operation simpler and more efficient.

After splitting the code between the two programs and adding a communication mechanism, all of the data structures required by the serial SA were identified. Like most large FORTRAN programs, PISCES data structures are defined and passed via COMMON blocks. It was a relatively simple task to identify all of the COMMON blocks needed by the serial SA. Preceding computation, the UI performs a wholesale transfer of necessary COMMON blocks to the serial SA using the communication routines (Figure 3.3). In this fashion the data transfer is isolated from and transparent to the original code.

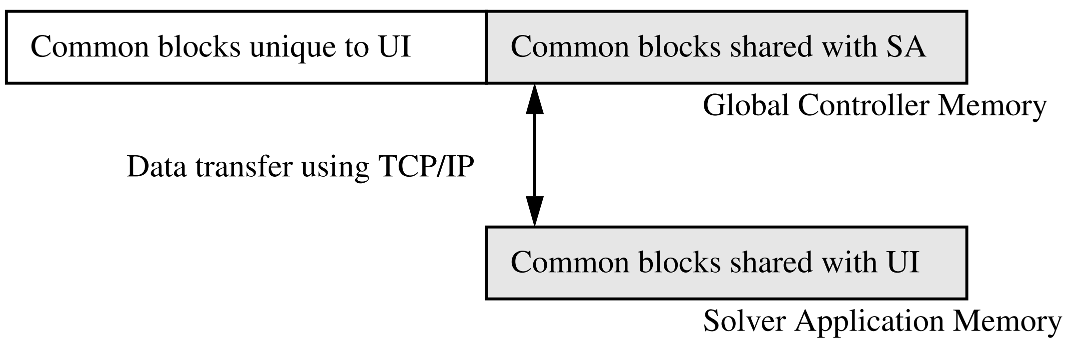

Figure 3.3 Data transfer using TCP/IP between UI and serial solver.

A coarse-grain choreographer to coordinate the transfer of data and solution commands was also necessary to complete the global controller. The coarse-grain commands consist of a series of bias conditions (each bias condition requires a separate nonlinear solution) to be solved by the SA. The UI choreographer interfaces with the main dispatch loop of PISCES (the routine which parses user's commands and calls the appropriate subroutines) and intercepts commands that require remote processing by the SA. The UI choreographer then transfers the data necessary for the computation requested followed by the actual command. It awaits notification from the SA choreographer that the command has been executed and receives the results. At the other end, the SA choreographer waits to receive messages, initiates the requested action, and returns the results. The changes to the original code were minimal since the choreographer is entirely external to both the UI and the SA. Only the dispatch loops were modified to interface with the choreographer. The resulting global structure of the simulator is shown in Figure 3.4.

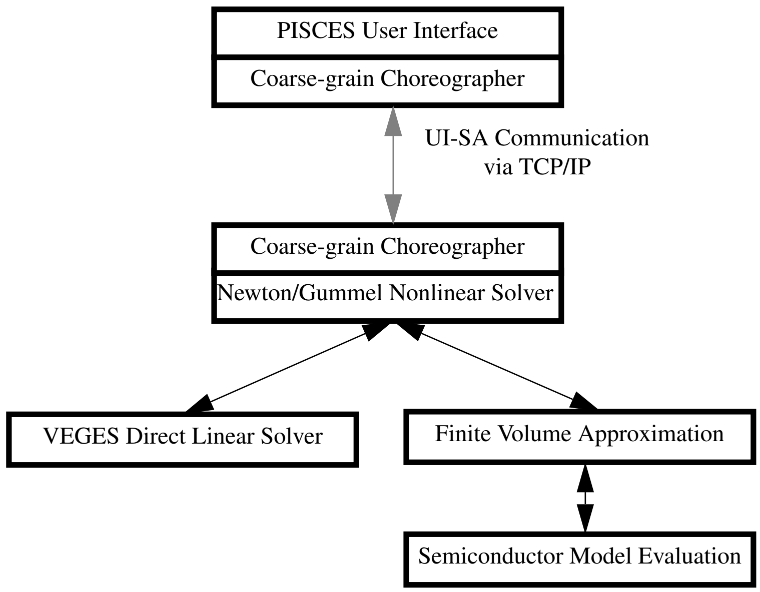

Figure 3.4 PISCES decomposition: UI/SA program structure.

3.2.2.2 Adding Domain Decomposition

Parallelism in the data is exposed through grid decomposition. Naturally, the first task for application parallelization is to add a domain decomposition module (DDM) to the global controller. The goal of the DDM is to divide the computation equally among the processors while minimizing the communication by keeping boundaries small. The creation of well balanced domains is difficult since these two goals are often in conflict. Moreover, an efficient decomposition for one phase of computation (e.g., model evaluation) may be less efficient for others phases (e.g., linear solution). The use of a single decomposition throughout the computation may therefore lead to some performance degradation. On the other hand, the cost of computing and maintaining multiple decompositions as well as the communication cost of shuffling data among processors between computational phases may offset the gains offered by better decomposition. Given the questionable performance gains offered by maintaining multiple decompositions and the added code complexity introduced, PISCES-MP was designed to use a single decomposition throughout all phases of computation. At runtime, the user's grid structure is given to the DDM which then divides the grid among the processors. The changes to the UI code were minor: a call to the DDM after the grid structure is loaded produces the partitioning.

Grid partitioning is an active area of research and has produced many robust, general-purpose tools in recent years [37]. One such method, the recursive spectral bisection (RSB) algorithm of Pothen *et. al.* [42] and Simon [49] was chosen as the heart of PISCES-MP's DDM. The DDM consists of the RSB code, a pre-processor to convert a PISCES grid description into an RSB grid description, and a post-processor to correct any inconsistencies in the partitioning. RSB employs a graph-theoretic approach to domain decomposition. The algorithm consist of three steps:

1. For a given mesh with n vertices, an n by n Laplacian matrix, Q, can be constructed using the adjacency structure of the mesh. The Laplacian matrix Q is defined by:

$$q_{ij} = \begin{cases} -1 & \text{if an edge exists between i and j} \\ 0 & \text{if no edge exists between i and j} \\ deg(i) & \text{for i=j} \end{cases}$$

2. Compute the eigenvalues of Q and order them as $\lambda_1 = 0 \le \lambda_2 \le \ldots \le \lambda_n$. Compute the second eigenvector corresponding to the smallest positive eigenvalue.

3. The components of the second eigenvector computed in Step 2 are used to separate the mesh vertices. All vertices with eigenvector components above the median value of the second eigenvector are put into one partition while those below are put into a second partition. A vertex separator is then computed using a matching algorithm.

Subsets produced by these three steps are recursively divided until the desired number of partitions (normally a power of two) has been reached. The most expensive step in the algorithm is the computation of the second eigenvector. The recent introduction of a multilevel implementation [29] of the construction of the Laplacian matrix has reduced the DDM's computation time to the degree that it is an insignificant part of the overall runtime for most simulations. However, the memory consumption of the RSB algorithm may render it unusable for very large grids.

To decompose grids too large for RSB, a low-memory solution was also implemented in PISCES-MP. The low-memory approach uses incomplete nested dissection (IND) based upon the level-sets of the mesh [19]. This algorithm bases its decomposition strictly on the connectivity of the underlying mesh. Level sets are constructed by a depth-first search of the mesh as follows:

1. Select a set of starting nodes and put them into level set#0. Mark the nodes as inactive. Let n=0.
2. Put all active neighbors of nodes in the *n*th level set into the *n+1*st level set. Mark the nodes as inactive.
3. Let n=n+1. If active nodes remain, repeat Step 2, else quit.

Figure 3.5 shows a level-set structure imposed upon a simple rectangular grid. Removal of any level set will divide the mesh into two disjoint pieces. The level set which produces the most evenly balanced partitions is usually chosen as the separator. IND may produce less well-balanced partitions, but consumes far less memory than RSB, making it suitable for very large meshes.

The domain decomposition algorithms described above are designed to handle homogenous grids (i.e., all nodes in the grid have equal characteristics). The addition of boundary conditions in the form of external electrodes (Figure 3.6) can create inconsistencies which must be corrected by post-processing the partitions. The electrodes are included in the grid description passed to the DDM; however, they are different from other nodes in the grid since they affect the assembly of all nodes connected to them. If the set of nodes attached to an electrode is spread across multiple processors by the domain decomposition algorithm (Figure 3.7), irregularities will occur if the electrode is not designated as shared among the processors. This can easily occur since the DDM cannot guarantee that an electrode will be divided among all partitions containing attached nodes.

The post-processor examines the partitioning and insures that all processors involved with nodes attached to an electrode are also involved with that electrode.The result of applying this operation causes electrodes to be shared across all involved processors. (Figure 3.8) and the boundary information to be distributed properly.

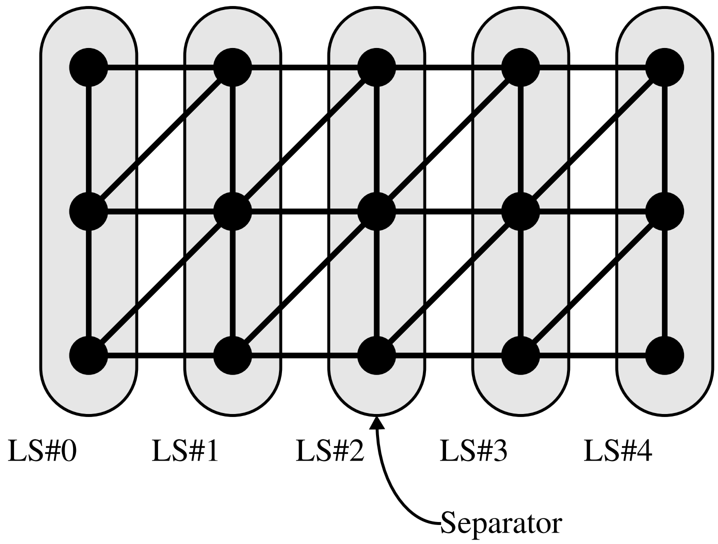

Figure 3.5 Example of level-set dissection.

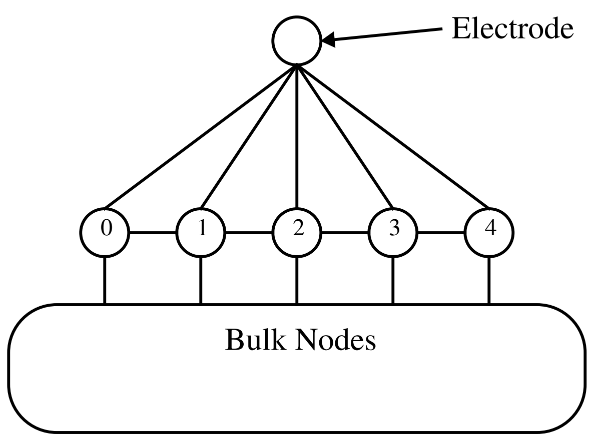

Figure 3.6 Electrode attached to 5 grid nodes.

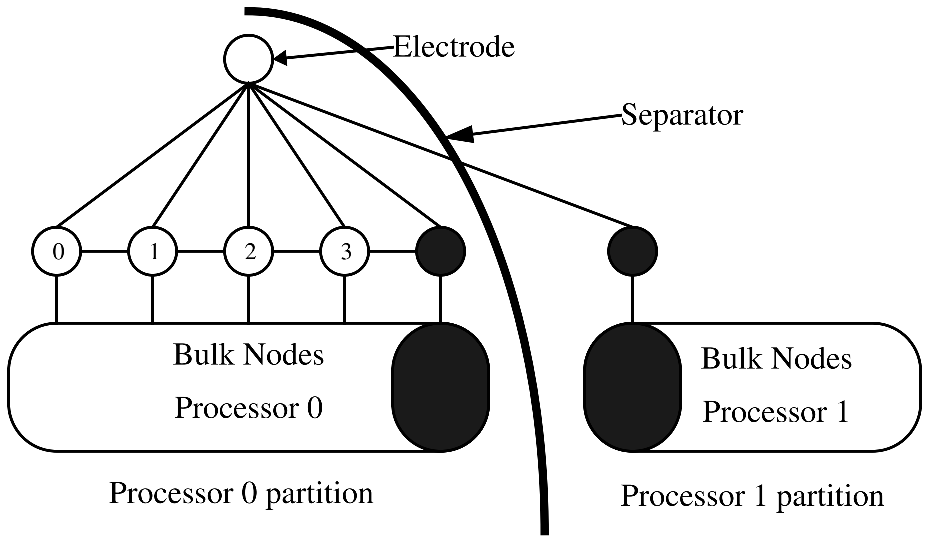

Figure 3.7 Initial RSB partitioning: the electrode has not been distributed among all processors that require its data.

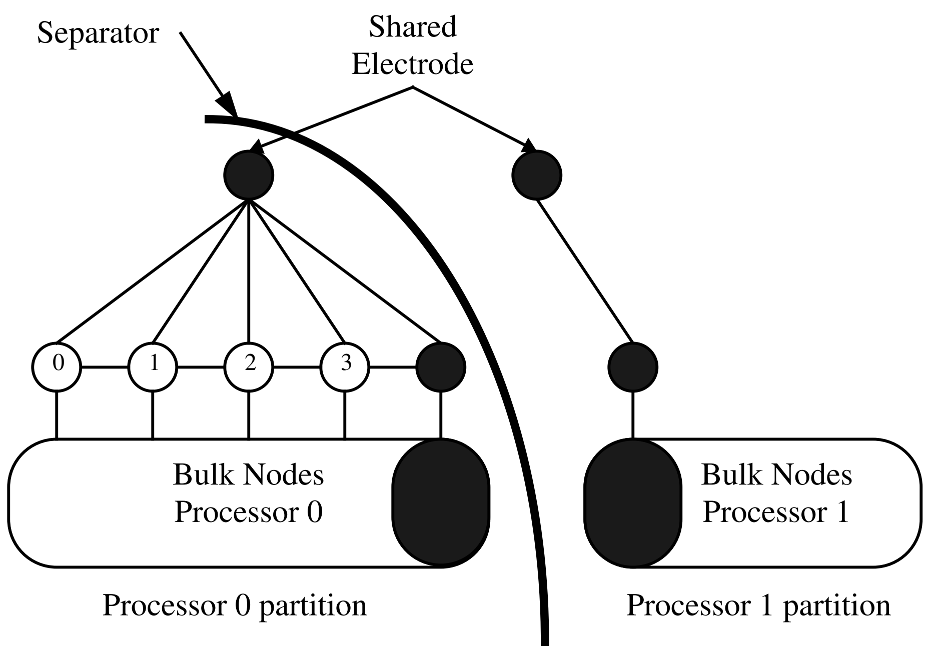

Figure 3.8 Corrected partitioning: the electrode has been distributed to both processors.

3.2.2.3 Parallel Data Distribution and Control

Once the global controller has the ability to partition grids for parallel solution, it must be modified to send data to all processors in the parallel partition. The global controller continues to pass information to the solver via FORTRAN COMMON blocks. In the parallel implementation, each processor only receives a portion of each data block pertaining to the mesh points in its local partition. Due to memory constraints, the global controller computes and sends partitions one at a time. Figure 3.9 shows the simple algorithm employed to create each partition. For each partition to be created, the global controller examines the domain and adds mesh points residing in the current partition to a temporary buffer (Figure 3.10). Once the partition has been created, the communication library is used to send it to *node_0*. Since it would be difficult to manage a separate TCP/IP connection to each processor (some machines have several hundred or more), the global controller continues to connect to only one processor, *node_0*. Data to other nodes is sent through *node_0* and forwarded to the destination via the parallel message passing library. The Node Executive (NX), Intel's message passing library [62], is much faster than the TCP/IP connection making the extra hop through *node_0* relatively inexpensive. Once data arrives at the destination, it is loaded into the corresponding COMMON blocks using the existing code.

```
for processor = 1 to number_of_processors
    for i = 1 to number_of_grid_points
        if (processor involved_with_node(i))
            add(node(i), partition(processor))
        endif
    endfor
    send_data(partition(processor))
endfor
```

Figure 3.9 Pseudo-code for partition creation.

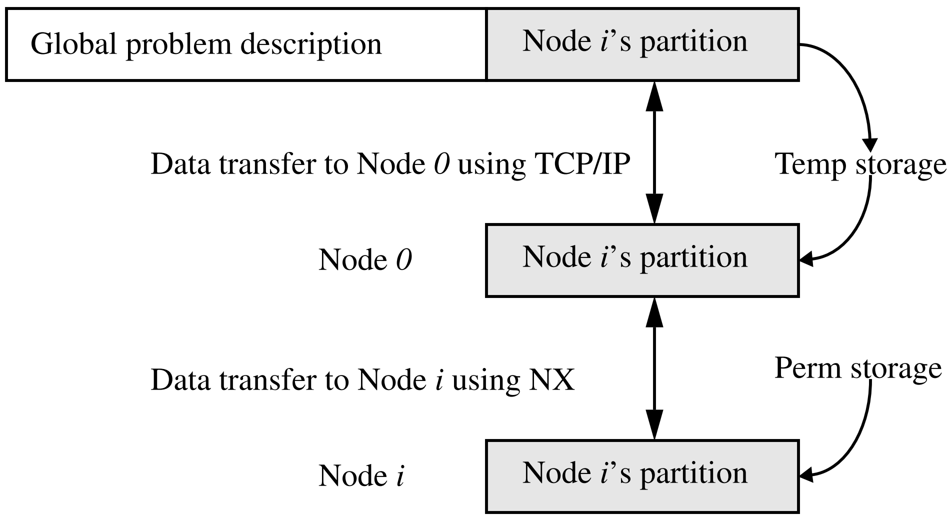

Figure 3.10 Two-step data transfer using TCP/IP between UI and serial solver and NX within the parallel computer.

Once the data is distributed, computation can be parallelized. Each processor runs a copy of the SA on a subgrid, communicating with other processors to resolve data dependencies. To manage this communication and synchronization a fine-grain choreographer which acts as the local controller coordinates the processors much in the same fashion as the coarse-grain choreographer coordinates actions between the UI and SA. *Node_0* oversees the parallel action by sending cues the other nodes to initiate communication, synchronization, and computation. The structure of the parallel simulator is shown in Figure 3.11.

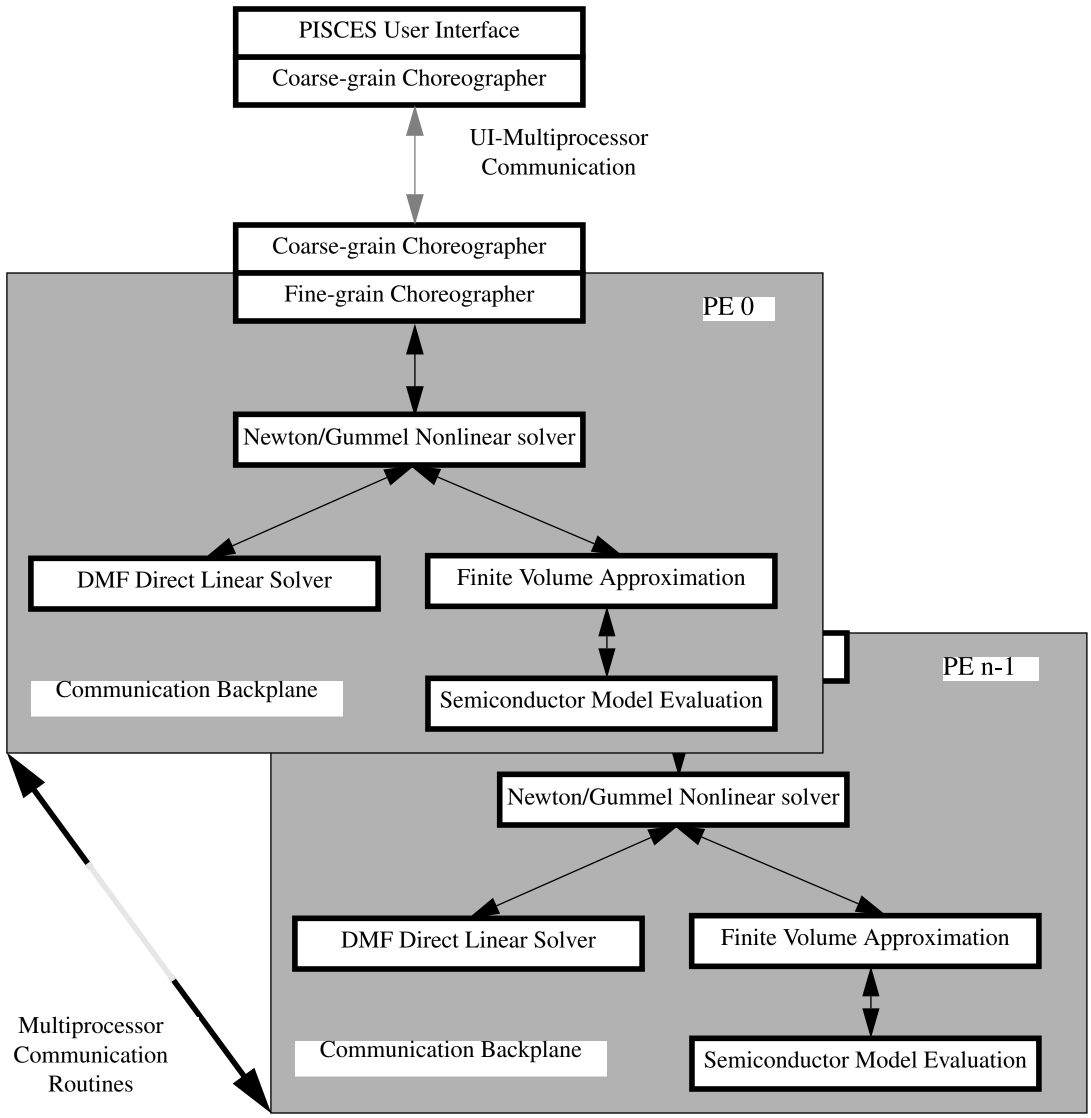

Figure 3.11 Global structure of a parallel simulator. The fine-grain choreographer controls execution and performs all communication and synchronization.

3.2.2.4 Parallel Communication Management

To manage communication during the parallel solution of the PDEs, a master-slave scheme is used. The master-slave scheme assigns a master processor to each shared data item. All other processors sharing this data are considered slaves of the master processor. The task of the master processor is to manage the shared data and insure that a consistent view of the data exists on all processors. After grid decomposition, the global controller performs the master and slave assignments of the processors and equitably divides the work among them.

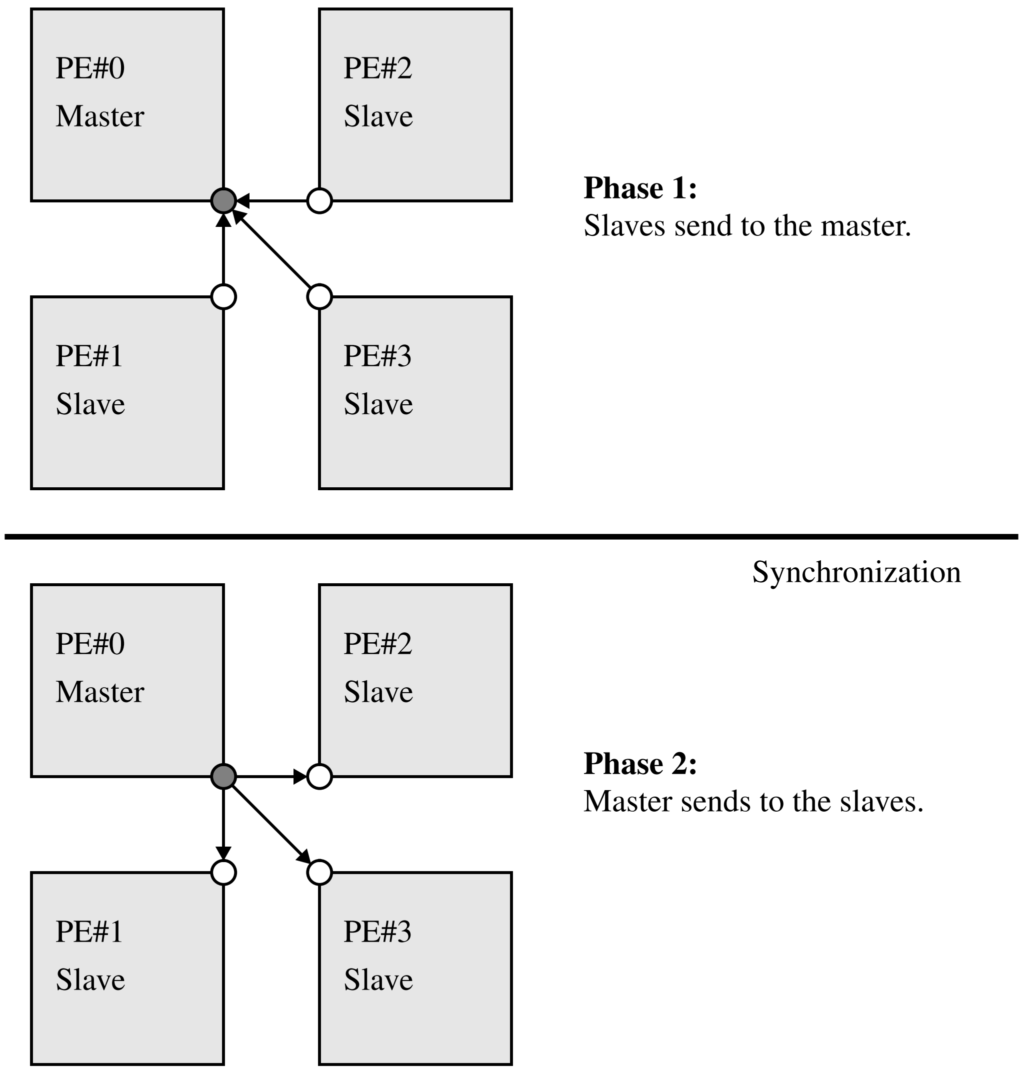

Figure 3.12 Two-phase vector globalization using master-slave assignments.

Typically, a two-step algorithm is employed during master-slave communication. The globalization of nonlinear residuals is one example of this approach. Figure 3.12 shows the globalization of a mesh point shared among four processors. In the first phase of globalization, all slave nodes send their contributions to the global value of the shared node to the master. After a synchronization, the master sums all contributions to create the global value. In the second phase of vector globalization, the master sends the result back to the slave nodes. In turn, the slaves reset the shared nodal value with the consistent global value. A similar approach is implemented for other shared data types.

3.2.2.5 Nonlinear Solver and Initial Guess

The nonlinear solver creates an initial guess at the solution and then repeatedly asks for a new linearized matrix and residual vector to be formed, requests the linear solution of this system, and applies the resulting nonlinear update until the convergence criteria are met. In the PISCES nonlinear solver module, only the initial guess routines require parallel communication. The matrix formation routines, residual globalization, and the linear solver reside in separate modules and will be discussed in later sections.

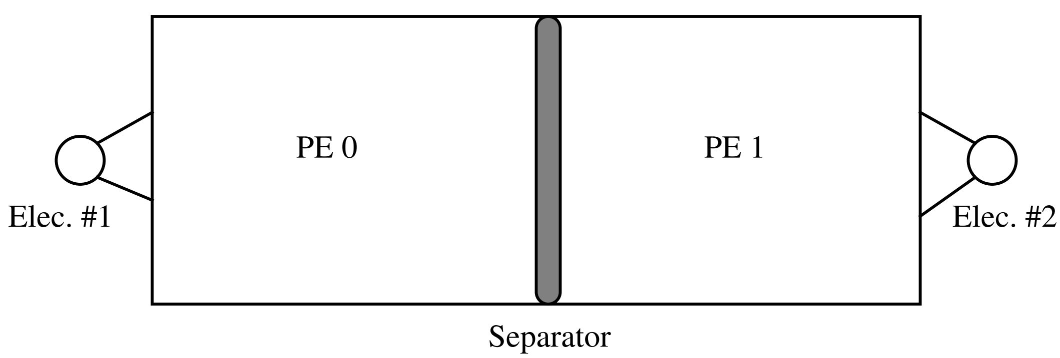

Figure 3.13 Non-locality in nonlinear projections.

The initial guess routines need non-local information when creating a new guess at the solution based upon bias conditions. Once again, the boundary conditions create inconsistencies. Two types of problems occur; one is solved through communication, the other through replication. The first case (shown in Figure 3.8) occurs when nodes tied together by a boundary condition are spread among several processors. To insure proper behavior, the attributes of all attached nodes must be examined and agreed upon globally. As these attributes are constantly changing and the number of involved nodes may be large, the only viable solution is to add a communication step during this phase of the initial guess algorithm. In the second case (shown in Figure 3.13), to compute the extrapolation factor for a projection properly, all electrodes (but not their attached nodes) must be examined. The projection is scaled by the differences between the new bias conditions and the previous values. However, the bias conditions are now distributed among many partitions and no processor can be guaranteed to contain a global view of the boundary conditions. To obtain this global view, two possible solutions were evaluated:

- Add a communication phase to the algorithm to make all processors agree upon the correct extrapolation factors. This approach has the disadvantages of adding communication overhead and requiring a moderate amount of code modification.
- Add extra data structures, replicated on each processor, which contain global boundary condition information. This has the disadvantage of requiring additional data structures to store the duplicated global information. It has the advantages of requiring no communication and minimizing the code changes. (Simply modify the call to the projection algorithm so that the global, rather than the local, boundary conditions are passed.)

The latter solution was implemented for three reasons: existing data structures were unaffected, the code modifications were minimal, and the amount of replicated data was small and did not impact memory requirements measurably.

3.2.2.6 Matrix Assembly and Physical Models

The assembly process proceeds in a finite element fashion, evaluating on a triangle-by-triangle basis. Each triangle forms its local contribution which is then added to the global system. If all information needed during assembly is local to a triangle, assembly may take place independently on all processors. This is usually the case; however, some of the physical models do require non-local data. Due to the large amount of volatile information involved, there is no simple way to avoid communicating to resolve non-local references.

Semiconductor devices often contain different materials within a single structure. One example is a silicon-oxide interface (Figure 3.14) common in MOS structures [38]. The mobility of the carriers along these interfaces differs from the mobility in bulk silicon. If the user desires to model this mobility variation, each silicon triangle along the interface must have knowledge of the material type and present electrical parameters of its neighbors in order to properly compute mobility at the interface. This model was modified so that interface triangles could obtain the material parameters of neighboring triangles on remote processors. Before evaluating this mobility model, the local controller initiates a communication phase to exchange the data required for model evaluation. The model itself was modified to access a temporary buffer containing the non-local information. By performing all communication beforehand, code changes were again minimized. Fortunately, models of this type are rare in the current release version of PISCES.

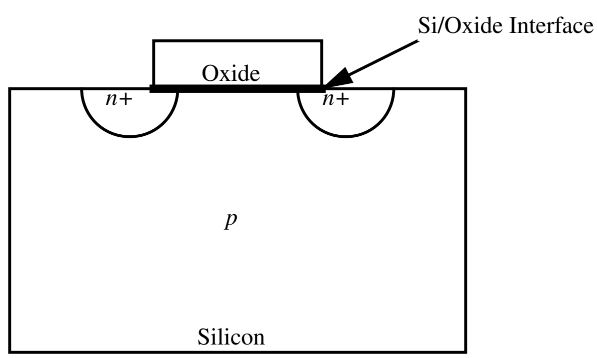

Figure 3.14 A simple *n*-channel MOS structure.

3.2.2.7 Linear Solution Decomposition

The linear solution code is distinct from the rest of the simulator and has a well-defined interface. The linear solver accepts a matrix and a right-hand-side vector from the nonlinear solver and returns an update vector. As long as the linear solution is correct and computed efficiently, the linear solver can be treated as a "black box" by the rest of the application. The serial version of PISCES uses a public domain sparse direct solver [5]. Although written more than fifteen years ago, it continues to be quite efficient on most serial architectures. However, adapting this code for parallel execution is problematic as the algorithms and data structures do not map well to modern parallel architectures. Fortunately, the segmentation between the linear solver and the remainder of the code allows one "black box" to be swapped for another one with identical behavior. The replacement of the linear solver had no effect on the remainder of the code.

The existing solver was replaced with a distributed multi-frontal (DMF) solver [9] modeled after the work of Lucas [18]. A detailed description of the algorithms and implementation of this solver will be given in Chapter 4 of this dissertation. The following discussion is intended to briefly describe the linear solver and provide insight into issues involved in its integration into the PISCES-MP application.

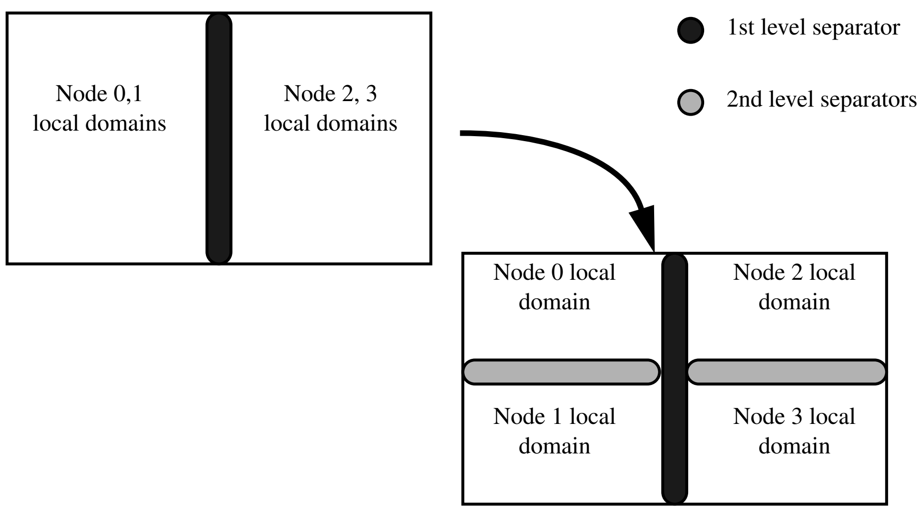

Figure 3.15 Creation of 4 partitions: The first separator (level 1) creates two partitions. The second set of separators (level 2) doubles the number of partitions by bisecting the previous partitions.

Sparse direct methods generally contain two phases: a symbolic factorization followed by a series of numeric factorizations. The symbolic phase first reorders the equations to minimize non-zero fill-in during elimination. This also minimizes the floating point operations needed to compute the factor and the memory required to store the factor. Once a reordering has been computed, the matrix is symbolically factored to the determine the non-zero structure of the factored matrix. Using this information, the requisite data structures are created. The numeric factorization performs the actual floating point work using the elimination ordering and storage prepared by the symbolic phase.

In parallel, the symbolic and numeric factorizations become more complicated due to both the constraints on elimination order posed by the separators and the communication required to factor them. The DDM creates partitions in a top-down fashion (Figure 3.15). This behavior is exploited when computing the order of elimination by factoring the local domains independently followed by the sets of separators. The sets of separators are factored in the inverse order in which they were created by the DDM (Figure 3.16).

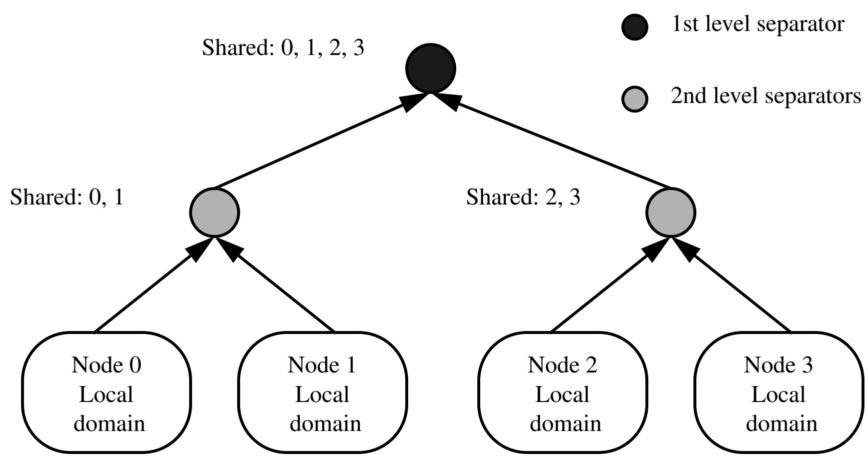

Figure 3.16 Direct linear solver data dependencies for a 4-processor partitioning. Information flows in a "bottom-up" fashion.

To accommodate the constrained order of elimination, a two-phase reordering is used. First, the multiple minimum degree (MMD) algorithm [19] is used to reorder each processor's local domain. Afterward, equations within the separators are added to the elimination ordering. The interiors are factored independently; but, the processors must communicate to factor the separators.

The non-zero fill-in during a sparse direct factorization adds coupling between equations residing on different processors. To insure a correct linear solution, processors must have knowledge of some equations that are resident on other machines. The global controller adds the extra grid points that generate these equations to the processors' data. These grid points are used solely by the linear solver and have no effect upon the remainder of PISCES.

3.2.2.8 Retrospective: Parallel Data Requirements

In the previous sections, the procedures followed for implementing the parallel PDE solver were described. To provide a complete picture of the parallelization process, it is useful to describe the major data structures added during development. These data structures are used by the choreographers. They are external to PISCES and have no effect upon the existing data structures. They are summarized below:

- *local_to_global_map*[1..#local grid points]: Each processor keeps an array which maps the local node numbering of its subgrid to the global node numbering. All communication among the processors uses the global numbering system.
- *owner*[1..#local grid points]: The array used to determine the processor in charge of accumulating updates to each shared grid point. The owner is responsible for communicating a consistent set of values for the grid point to other processors sharing the node.
- *share_count*[1..#local grid points]: This array, in conjunction with *share_list*, provides a complete picture of the interactions with other processors for each the shared grid point. The data is stored using a quotient list structure [10]. For each grid point, *share_count* points to the starting location in *share_list* of that node's list of involved processors (Figure 3.17). The number of processors sharing node i is given by *share_count*[i+1] - *share_count*[i].
- *share_list*[1..share_count[#points+1]]: This array (Figure 3.17) contains lists, one for each grid point, of the processors sharing a grid point. It is indexed by

share_count.

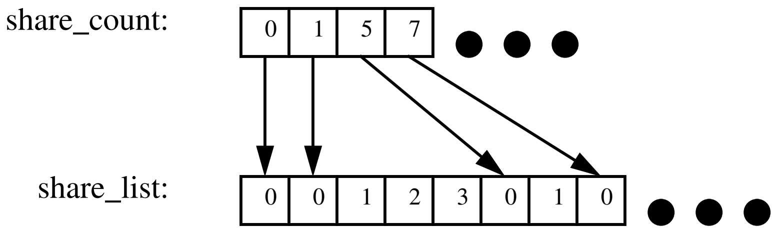

Figure 3.17 The share_list and share_count arrays provide a profile of shared node interactions.

- *separator_level*[1..#local grid points]: Tells which level separator a shared grid point belongs in. It is used exclusively by the linear solver as described earlier.
- *Global Electrode Data*: This is a complete set of boundary condition data for the global system to insure consistency across the processors as discussed above.
- *elem_owner*[1..#global triangles]: This array is used solely by the UI program to determine the proper destination for each triangle. It is not sent to the SA.

3.2.3 Porting Examples: CM-5 and SP1/2

To prove that the coarse-grain methodology creates applications that can be easily ported to other parallel computers supporting the message-passing model, PISCES-MP was moved to the CM-5. The porting process was surprisingly straightforward. Most of the work involved changing the UI/SA communication layer. Since the CM-5 has one (or more) SUN workstation equivalents as control processors, the UI was run on a control processor rather than a remote workstation. The control processors provided acceptable performance and a 20Mb/s connection to the attached parallel machine, a considerable improvement over TCP/IP communication. Also, the host can communicate directly with all nodes, making the intermediate hop through *node_0* unnecessary. The host uses the same byte ordering as the parallel processors, making byte reordering unnecessary.

Otherwise, NX calls were translated into corresponding CMMD calls. In most cases, a simple wrapper was sufficient and the existing code was left untouched. In rare cases, such as the hypercube-specific *cubedim()* function, functions more appropriate to the CM-5 were substituted. The porting work required less than two weeks and minimal code changes were necessary.

The port to the SP1 was also straightforward. The local ethernet of the SP1 was utilized, retaining the TCP/IP connection between the UI and the SA. Again, calls to NX were emulated by message-passing calls in MPL. However, difficulties arose when emulating non-blocking sends. The NX system buffers non-blocking messages allowing user buffers to the reused immediately after the send returns. MPL does not buffer non-blocking messages and forces the user program to wait until the receiver accepts the message before reusing its buffer. To avoid rewriting the existing code to account for this behavior, the behavior of NX was mimicked by copying messages to a neutral buffer within the emulation routines. Thus, the existing code can continue to reuse its buffers. To date, no significant performance impact using the approach has been observed. As with the CM-5E, the porting work required less than two weeks and minimal code changes were necessary.

3.3 PISCES-MP Results

Using the new parallel device simulator, the computational resources can be scaled to match problem requirements. This should provide not only the ability to simulate existing grids in less time but also the ability to simulate large, complex grids which are computationally infeasible on serial architectures. The results presented in the following sections demonstrate both capabilities. Results are given for the three architectures currently supporting PISCES-MP. To provide a baseline, runtimes on a several serial workstations are also provided. The performance of the fastest serial workstation available (RS/6000 Model 590) is used for comparisons. The other workstation runtimes are provided for reference. In most cases, results are presented as graphs of solution times. Actual solution times for these results may be found in Appendix A. In some cases (e.g., the CM-5) tabular form is more convenient. For tabular data, an entry of “N.A.” denotes that the data could not be obtained due to resource constraints (usually lack of memory). If single-processor performance is unobtainable, the smallest number of processors capable of running an example is used as a baseline to compute speedup.

3.3.1 PN Diode

One simple and widely used semiconductor device is the PN diode [38]. A simple simulation described in the PISCES users's manual [40] which determines the forward-bias characteristic of a PN diode was run for a mesh comprised of 10,000 grid points on a 25x400 grid. (Problem statistics are summarized in Table 3.1.) The simulation required over 45 minutes on the IBM RS6000/590. (Table 3.2 summarizes serial workstation runtimes.)

Table 3.1 PN diode grid statistics.

	10K Grid
Number of grid points	10,000
Number of triangles	21,223
Number of equations	30,000

Table 3.2 Serial solution times for the PN diode simulation.

	10k Grid
Sparc2	21,234s
RS6000/530H	8,151s
RS6000/560F	5,225s
RS6000/590	2,932s

Figure 3.18 compares the runtimes on the iPSC/860 and the RS6000/590. The iPSC/860 had insufficient memory to run the simulation with less than four processors. The first observation is that the RS6000/590 is competitive with four processors of the iPSC. This result is not unexpected since the RS6000/590 was introduced 4 years after the i860 processor of the hypercube. (The iPSC compares more favorably to the older, less powerful workstations.) The parallel code exhibits satisfactory speedups (versus four

processors) of 1.8x and 3x on 8 and 16 processors, respectively. However, only a 3.8x speedup (versus four processors) is achieved on 32 processors.

Remark: It has been observed over a wide range of problems that the efficiency of PISCES-MP suffers on both the iPSC and SP1/2 if the number of mesh points per processor falls below 500. This limitation will recur in later examples.

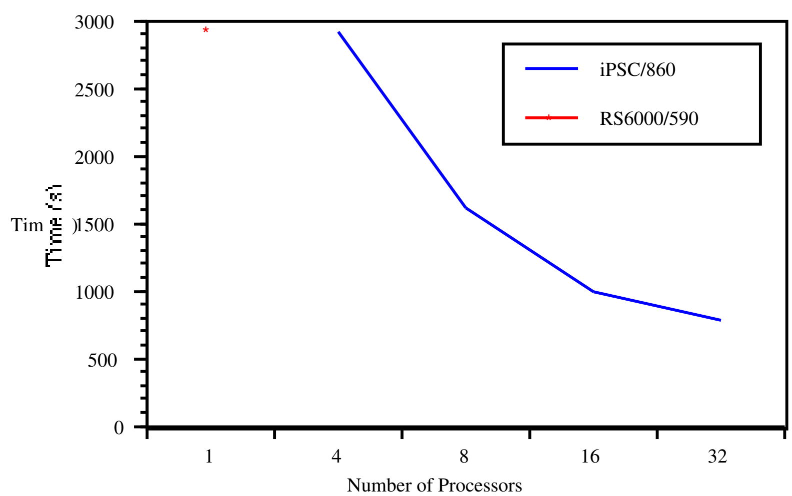

Figure 3.18 iPSC/860 and RS6000/590 solution times for PN diode.

Table 3.3 CM-5E solution times for the PN diode simulation.

	PN Diode Total Time
32PE	817s
64PE	499s

Table 3.3 summarizes the execution times for PN diode simulations on the CM-5E. (Due to the small number of data points, a tabular form is used.) Increasing the number of processors from 32 to 64 yields a 1.6 speedup. Considering the small number of grid

points per processor (less than 150 for 64 processors), this result is quite good. The superior efficiency for large number of processors versus the iPSC/860 is due primarily to the faster connection network of the CM-5E (described in Chapter 2).

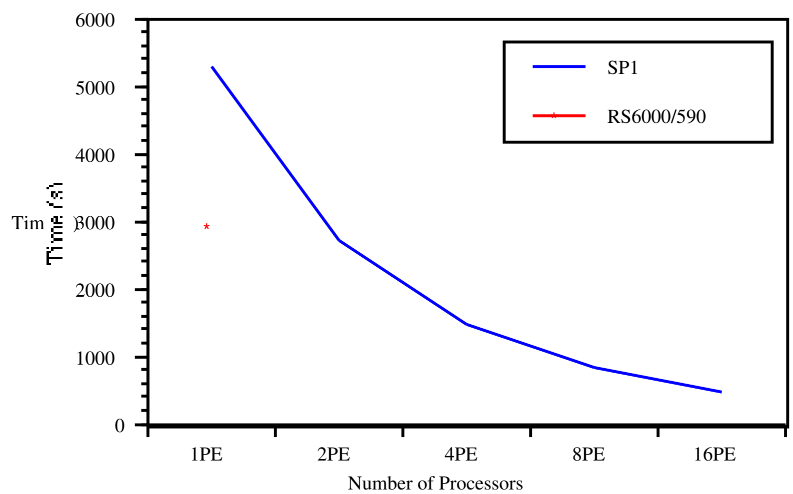

Figure 3.19 SP1 and RS6000/590 solution times for PN diode.

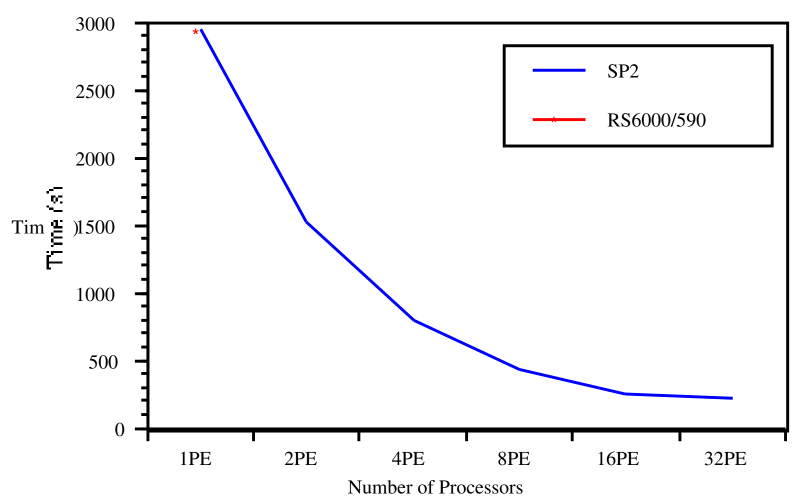

Figure 3.20 SP2 and RS6000/590 solution times for PN diode.

Figure 3.19 and Figure 3.20 show the runtimes PN diode simulation for the SP1 and SP2, respectively. The processors of the SP1 used in this study (RS6000/370) offer roughly one-half of the performance of the RS6000/590 workstation; thus, the SP1 compares favorably to the RS6000/590 and exceeds its performance at 2 processors. With 16 processors, the SP1 is over 6x faster than the RS6000/590. The SP1 achieves speedups (versus a single SP1 processor) of 1.9 on 2 processors, 3.6 on 4 processors, 6.4 on 8 processors, and 10.8 on 16 processors. Since the SP2 uses processors identical to the RS6000/590, it is natural that the single processor performance is identical to the RS6000/590. As more processors are added, speedups of 1.9 on 2 processors, 3.7 on 4 processors, 6.8 on 8 processors, 11.7 on 16 processors, and 13.3 on 32 processors are obtained. For up to 16 processors, scalability appears to be very good. At 32 nodes (like the iPSC/860) the amount of work per processor is insufficient to overcome the extra communication introduced.

3.3.2 Bipolar Transistor

Another popular semiconductor device in the bipolar transistor [38]. The structure used by Goossens, et.al. [23] to model the snapback characteristics of a trench-isolated bipolar transistor is typical of these devices. Although this mesh is slightly smaller than the PN diode mesh, the geometry is more complex and unstructured. Table 3.4 summarizes the simulation statistics. The snapback characterization simulation was run on the iPSC/860, SP1, SP2 and several serial workstations. Table 3.5 provides the serial workstation solution times.

Table 3.4 Bipolar transistor grid statistics.

	6K Grid
Number of grid points	5,931
Number of triangles	11,653
Number of equations	17,793

Table 3.5 Serial solution times for bipolar transistor simulation.

	6k Grid
Sparc2	107,153s
RS6000/530H	28,876s
RS6000/560F	19,849s
RS6000/590	9,716s

Figure 3.21 shows the iPSC/860 and the RS6000/590 runtimes of the bipolar simulation. Once again, the state-of-the-art RS6000/590 is equivalent to four or eight of the older i860 processors. At best, the iPSC/860 betters the RS6000/590 solution time by 3x using 32 processors. Computing speedups using the 4-processor time as a baseline, the iPSC/860 achieves speedups of 1.8x on 8 processors, 2.9x on 16 processors, and 3.4 on 32 processors. At both 16 and 32 processors, the workload is less than 500 mesh points per processor and is insufficient to justify the added communication overhead.

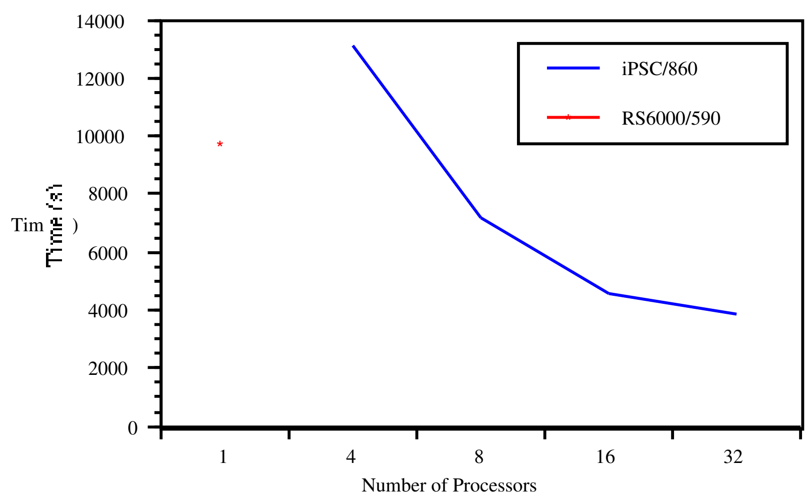

Figure 3.21 iPSC/860 and RS6000/590 solution times for bipolar transistor.

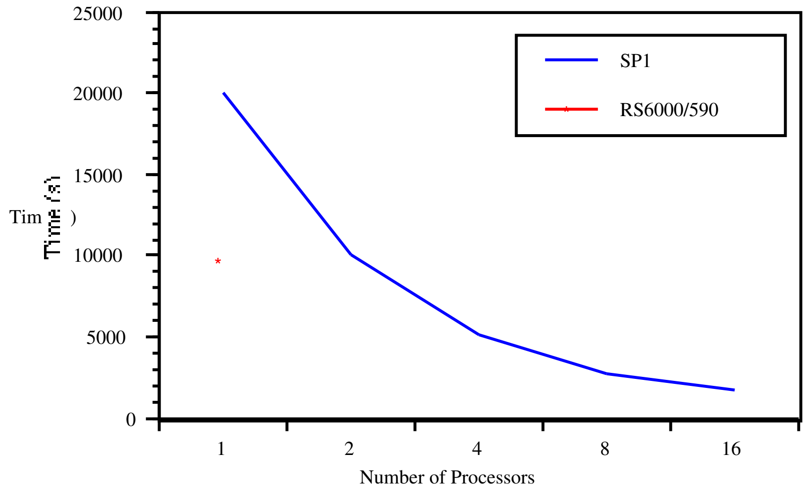

Figure 3.22 SP1 and RS6000/590 solution times for bipolar transistor.

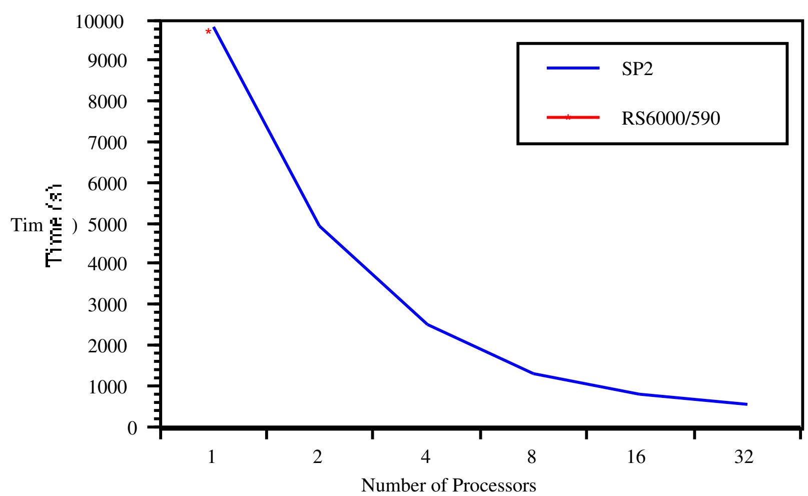

Figure 3.23 SP2 and RS6000/590 solution times for bipolar transistor.

Figure 3.22 and Figure 3.23 show the bipolar simulation runtimes on the SP1 and SP2, respectively. The SP1 matches the performance of the more powerful RS6000/590 at two processors. Using the single-processor SP1 performance as a baseline, speedups of 2x on 2 processors, 3.9x on 4 processors, 7.2x on 8 processors, and 11.9x on 16 processors are obtained. The SP2 achieves speedups of 2x on 2 processors, 3.9x on 4 processors, 7.5x on 8 processors, 12.4x on 16 processors, 18.9x on 32 processors over a single-processor SP2. The SP1 and SP2 speedups represent parallel efficiencies of 75% and 78% for 16 processors, respectively. These results are excellent considering the small amount of work (less than 400 mesh points) per processor. At 32 processors, the SP2 maintains 59% efficiency despite the extremely small workload per processor.

3.3.3 CMOS Inverter

One motivation for the development of PISCES-MP was to provide the capability to model the interactions among multiple devices. This ability becomes crucial as device dimensions are reduced. One multi-device structure of interest is a two-transistor CMOS inverter. The steady-state characteristics of a CMOS inverter were calculated using PISCES-MP on the four available parallel computing platforms. Four varying grid sizes, as described in Table 3.6, were simulated. They comprise a realistic set of grids to study various characteristics of this complex device. The lower resolutions are suitable to model the macroscopic behavior. The finer resolutions are required to capture small-scale effects within the device. The multiple grid resolutions facilitate the study of another aspect of PISCES-MP behavior: scalability with grid size. As the sizes of simulations grow, the computational resources of traditional serial architectures become overwhelmed. (Table 3.7 summarizes serial workstation runtimes for the various grid sizes.) However, by scaling computational resources to match problem requirements, PISCES-MP should continue to provide acceptably short runtimes as mesh sizes increase.

Table 3.6 CMOS inverter grid statistics.

	2K Grid	8K Grid	11K Grid	18K grid
Number of grid points	1,607	7,844	10,728	18,503
Number of triangles	2,963	15,158	20,873	36,400
Number of equations	4,821	23,532	32,184	55,509

Table 3.7 Serial solution times for CMOS inverter simulations.

	2K Grid	8K Grid	11K Grid	18K Grid
SUN Sparc2	1,214s	40,032s	95,433s	N.A.
IBM RS/6000 Model 530H	417s	11,263s	23,822s	46,017s
IBM RS/6000 Model 560F	348s	4,646s	15,687s	30,202s
IBM RS/6000 Model 590	181s	1,325s	2,253s	4,707s

Figure 3.24 shows the iPSC/860 runtimes for the 2K grid, the 8K grid, and the 11K grid. (The iPSC/860 had insufficient per-node memory to solve the 18K grid.) For the 2K grid, speedups are poor for more than 4 processors. In fact, the execution time for 32 processors is longer than the execution time for 16 processors. For example, using only 8 processors means the each processor's partition is less than 250 mesh points (the grid contains roughly 1,600 points). Clearly, this grid provides insufficient work to justify adding more processors. The two larger grids are slightly more promising. Since the iPSC/860 had insufficient per-processor memory to run these grid on less than 8 processors, the 8 processor result is used as a baseline for computing speedups. For the 8K grid, using 16 processors provides a 1.7x improvement over 8 processors, but 32 processors shows little improvement over 16 (again, too little work per processor). For the 11K grid, scaling to 16 and then 32 processors provides a steady incremental improvement in execution times.

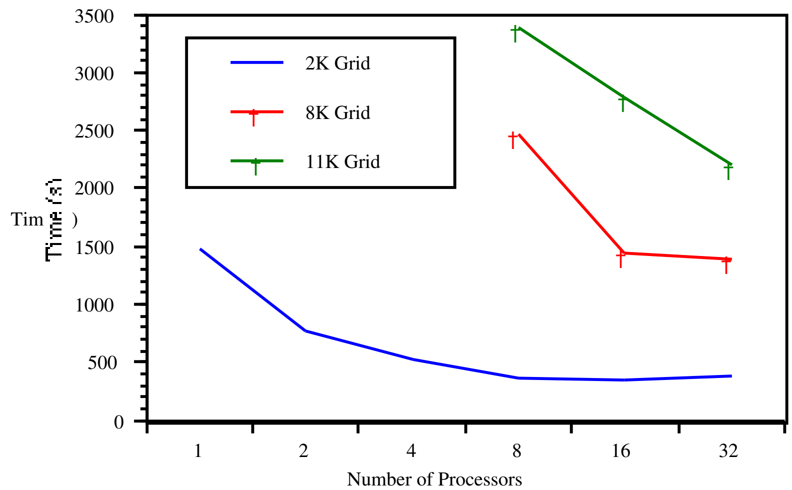

Figure 3.24 Intel iPSC/860 solution times for three CMOS inverter mesh sizes.

All four simulation grids were run on both a 32-processor CM-5E and a 64-processor CM-5E (Table 3.8). For the smallest grid, solution times were equivalent on both machines. The minuscule amount of work per processor is responsible for this behavior. For each of the three large grids, the 64-processor CM-5E provides an improvement of roughly 1.5x over the 32-processor machine. This moderate speedup is encouraging; because even for the largest grid, the 64-processor machine has less than 300 grid points per processor.

Table 3.8 CM-5E solution times for CMOS inverter simulations.

	2K Grid	8K Grid	11K Grid	18K Grid
CM-5E 32PE	262s	1,150s	1,792s	3,054s
CM-5E 64PE	270s	832s	1,232s	2,094s

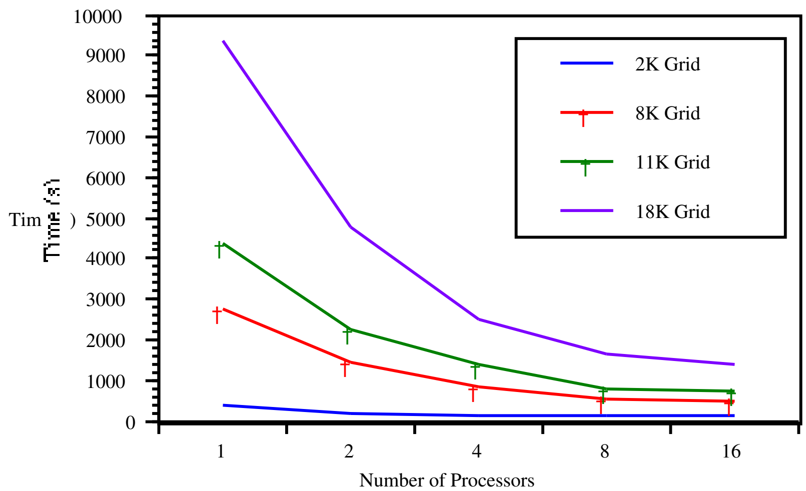

Figure 3.25 SP1 solution times for four CMOS inverter mesh sizes.

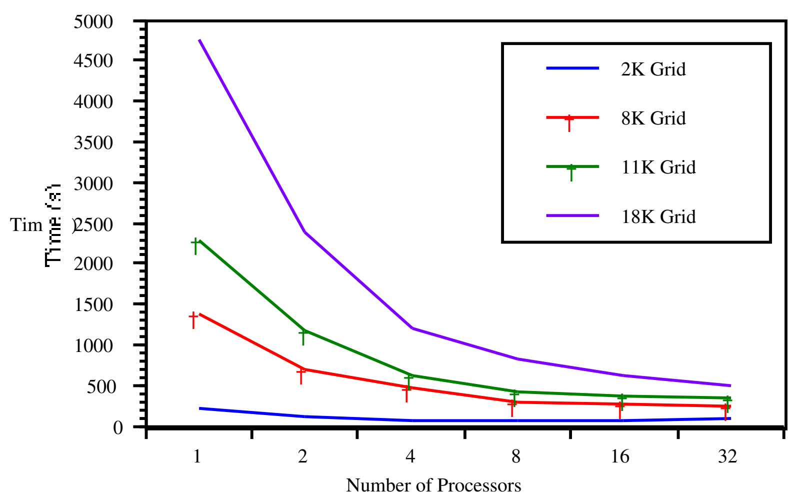

Figure 3.26 SP2 solution times for four CMOS inverter mesh sizes.

Figure 3.25 and Figure 3.26 show the four simulation runtimes on the SP1 and SP2, respectively. For the 2K grid, both machines exhibit behavior similar to the iPSC/860 with very poor efficiencies and increasing runtimes for more than a few (4 to 8) processors. On the 8K grid, the 11K grid and the 18K grid, the SP1 continues to provide good speedups (5.1x, 5.4x and 5.7x, respectively) on 8 processors, but begins to level off (5.5x, 5.8x and 6.7x, respectively) at 16 processors when the workload per processor has become too small. The SP2 exhibits similar but somewhat better behavior on the three larger grids due to its faster processor communication speed.

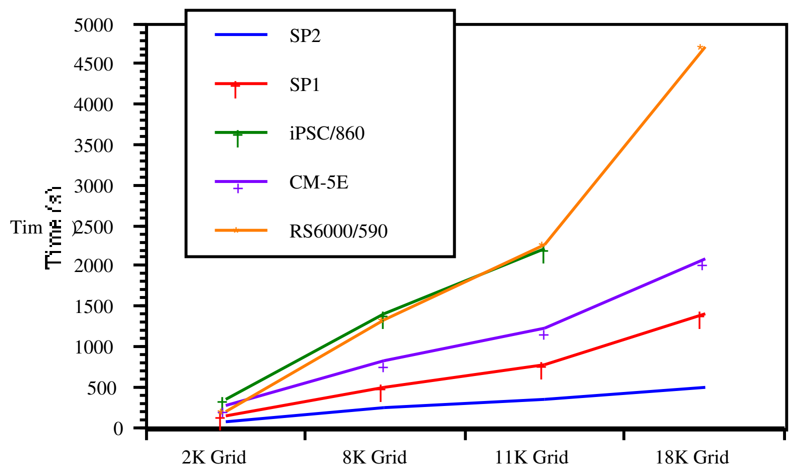

Figure 3.27 Solution time versus mesh size for CMOS inverter.

Figure 3.27 compares runtimes of the parallel computers and the RS6000/590 as mesh sizes increase. (Data shown is from a 32-processor iPSC, a 32-processor iPSC, a 16-processor SP1, and a 16-processor SP2.) For the 2K grid, the parallel computers cannot significantly outperform the RS6000/590. This first observation begs the question: why develop a parallel simulator? The answer is: to provide the ability to solve existing large problems quickly and the ability to solve larger problems which are computationally infeasible on current serial machines. For small problems, the overhead of setup, communication, and synchronization may make some simulations run slower in parallel than serially. The 2K grid is such a problem. However, the 8K grid is more encouraging. The SP2 (with its comparable processors) achieves greater than 5x speedup over the RS6000/590. The 11K grid is more reassuring; the SP2 provides a 7x speedup. On the 18K grid, the SP2 provides almost an order-of magnitude improvement over the RS6000/590.

3.3.4 Photo-Emitter

Another motivation for the development of PISCES-MP was to provide the capability to simulate devices and structures too large for most existing serial computers. One example of a device requiring immense amounts of grid arises in the study of atomic-scale devices such as hetero-junction photo-emitters [53]. These structures are comprised hundreds of layers of different materials. The layers are stacked hundreds deep and each layer is only a few (possibly only one) atomic layers deep. Extremely fine grid resolution is required to determine the behavior of the device. Table 3.9 lists the statistics for several different grid resolutions describing a sample photo-emitter. This device was simulated by PISCES-MP using simplified material models. Only the smallest grid could be solved using a workstation (Table 3.10).

Table 3.9 Photo-emitter simulation grid statistics.

	6K Grid	56K Grid	112K Grid	512K grid
Number of grid points	5,600	56,000	112,000	512,000
Number of triangles	10,467	108,551	218,423	1,001,027
Number of equations	16,800	168,000	336,000	1,536,000

Table 3.10 Serial solution times for photo-emitter simulation.

	6K Grid
Sparc2	62,558s
RS6000/530H	17,998s
RS6000/560F	10,283s
RS6000/590	3,007s

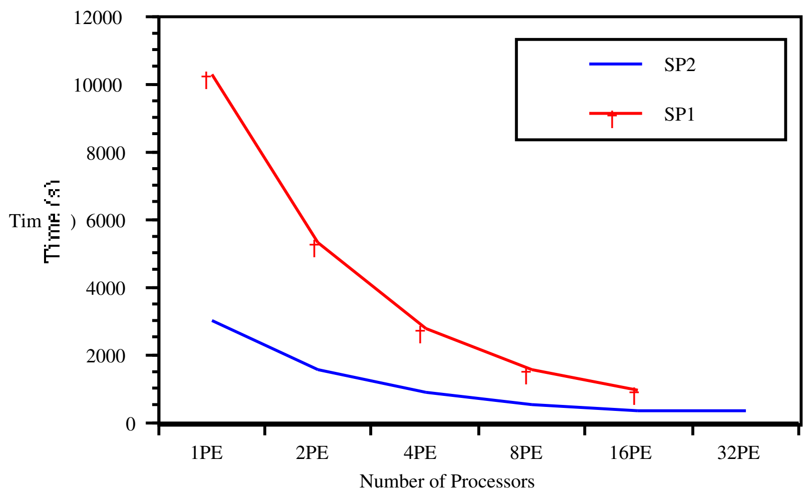

Figure 3.28 SP1 and SP2 runtimes for 5,600 grid-point photo-emitter.

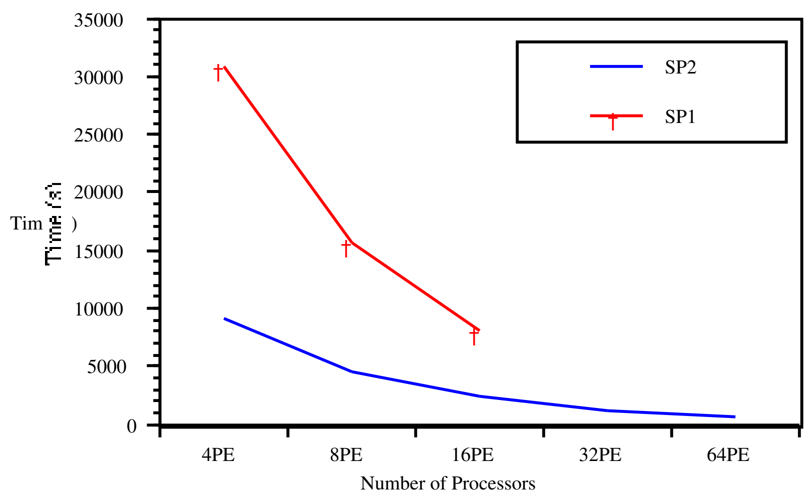

Figure 3.29 SP1 and SP2 runtimes for 56,000 grid-point photo-emitter.

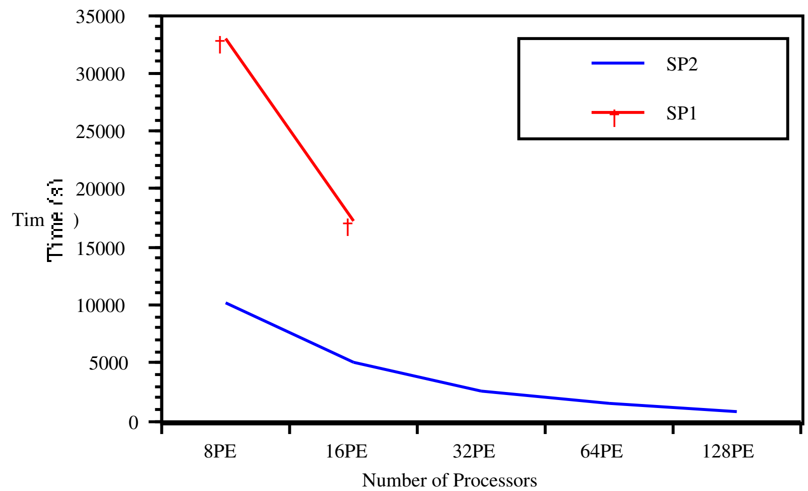

Figure 3.30 SP1 and SP2 runtimes for 112,000 grid-point photo-emitter.

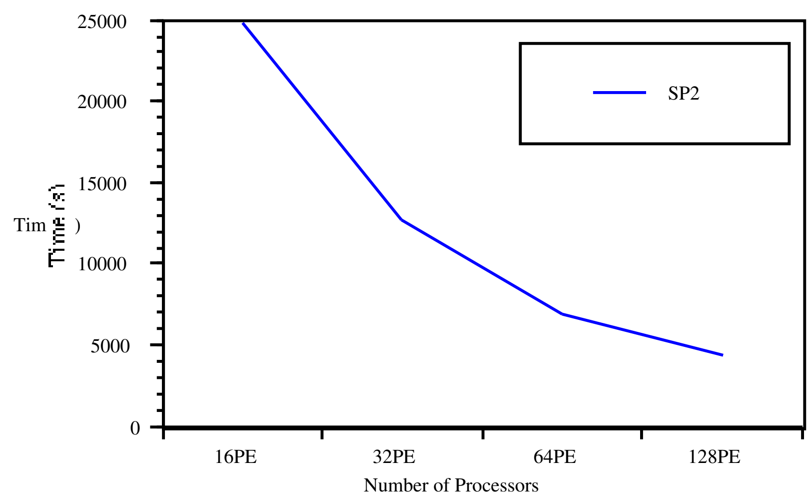

Figure 3.31 SP2 runtimes for 512,000 grid-point photo-emitter.

Figure 3.28 shows the runtimes of the SP1 and SP2 on the smallest grid (5,600 mesh points). The speedup for both machines is good through 8 processors (6.6x for the SP1 and 6.1x for the SP2) but begins to fall off at 16 processors (10.9x for the SP1 and 8.9x for the SP2) as the per-processor workload diminishes. The three larger grids are far too large to be solved on any available serial computer. Figure 3.29 shows the runtimes of the SP1 and SP2 on the 56,000 grid-point example. SP1 and SP2 configurations with less than four processors have insufficient memory to solve this example. The SP1 shows very good scalability for this problem with the 16-processor configuration finishing 3.8x faster than the baseline 4-processor configuration. Similarly, the SP2 performs well; 32 processors are 7.2x faster than 4 processors and 64 processors are 13x faster than 4 processors. Figure 3.30 shows the runtimes of the SP1 and SP2 on the 112,000 grid-point example. For this example, configurations of less than eight processors have insufficient memory. Using 16 SP1 nodes, yields a speedup of 1.9x on 8 processors. The SP2 achieves speedups of 2x on 16 processors, 3.9x on 32 processors, 7.1x on 64 processors, 11.3x on 128 processors, using the 8-processor runtimes as a baseline. Finally, Figure 3.31 shows the SP2 runtimes for the 512,000 grid-point example. Using the 16-processor number as a baseline, speedups of 2x on 32 processors, 3.6x on 64 processors, and 5.7x on 128 processors are achieved.

Remark: These results demonstrate the feasibility of solving “grand challenge” problems using coarse-grain parallel PDE applications on powerful parallel computers. The 512,000 grid point example required the solution of 1,536,000 simultaneous, non-symmetric equations. Previously intractable problems may now be solved in “engineering reasonable time” by parallel computers.

3.4 Framework Generality: FIESTA-HD

The adaption of PISCES for parallelism through the application of the coarse-grain model has not only created a parallel application but also a reusable control and communication framework which may be employed in the coarse-grain parallelization of other PDE codes. This section briefly discusses the parallelization of a second semiconductor device simulator (FIESTA-HD), highlighting the generality and reusability of the coarse-grain control and communication modules. A brief results section once again shows the potential benefits provided by employing the coarse-grain parallel model for PDE application parallelization. A full discussion of FIESTA and its parallelization is beyond the scope of this dissertation; for a full description of the application, see Aluru's thesis [1].

3.4.1 FIESTA-HD Description

As semiconductor devices are scaled to submicron sizes, more accurate models are needed to correctly capture device behavior. Historically, the additional computational overhead required by these more accurate methods made them impractical on serial computers. However, parallel computers can be employed to meet the computational demands of these methods. FIESTA-HD is one simulator which provides highly accurate models. FIESTA-HD solves the state-of-the-art hydrodynamic device equations employing a space-time Galerkin/Least-Squares finite element method. FIESTA-HD was built upon an existing finite element solver for compressible Euler and the Navier-Stokes equations. The linear solvers currently employed are GMRES to solve the non-symmetric hydrodynamic equations and conjugate-gradient for the symmetric Poisson equation.

3.4.2 FIESTA Parallelization

Unlike PISCES, FIESTA-HD is a well-engineered code and required very little restructuring for parallel execution. The global controller created for PISCES-MP, with its communication and domain decomposition modules, was reused for the parallelization of FIESTA-HD. The most time-consuming task was customizing the global controller for the data structures and user commands within FIESTA-HD. The finite element approximation employed by FIESTA-HD used purely local physical models and required no communication for parallelization. In fact, the parallel communication was limited entirely to the linear solvers. The remainder of the work involved adapting the iterative linear solvers for parallel execution by adding dot-product and vector globalization steps where necessary. As with PISCES, modifications to the existing serial code and data structures were minimal.

Initial parallel development took place on a 32-node Intel iPSC/860. FIESTA-HD was later ported to the Intel Touchstone Delta and the IBM SP1. These implementations were easily managed by the same porting techniques used for PISCES-MP.

3.4.3 FIESTA-HD Results

FIESTA-HD provides excellent performance both as problem sizes increase and as more processors are used for a fixed problem size. Figure 3.32 shows the runtimes of FIESTA-HD on an IBM RS6000/530H workstation, an 8-processor SP1, and a 32-processor iPSC/860 for several grid sizes. As the grid sizes increase, the parallel computers provide increasingly better solution times than the RS6000/530H. Not surprisingly, the SP1 (with its much faster processors) outperforms the iPSC/860. Figure 3.33 compares the runtimes of the iPSC/860 and the SP1 for an n-channel MOSFET comprised on 6,000 grid points. (The grid cannot fit on less the 4 processors of the iPSC/860.) Using the 4 processors as a baseline, the iPSC achieves speedups of 1.8x on 8 processors, 3.6x on 16 processors, and 5.4x on 32 processors. The SP1 achieves speedups of 1.9x on 2 processors, 3.1x on 4 processors, and 4.6x on 8 processors. As the results have demonstrated, the ability to scale computational resources to match problem requirements has made the use of the accurate but computationally demanding models of FIESTA-HD feasible.

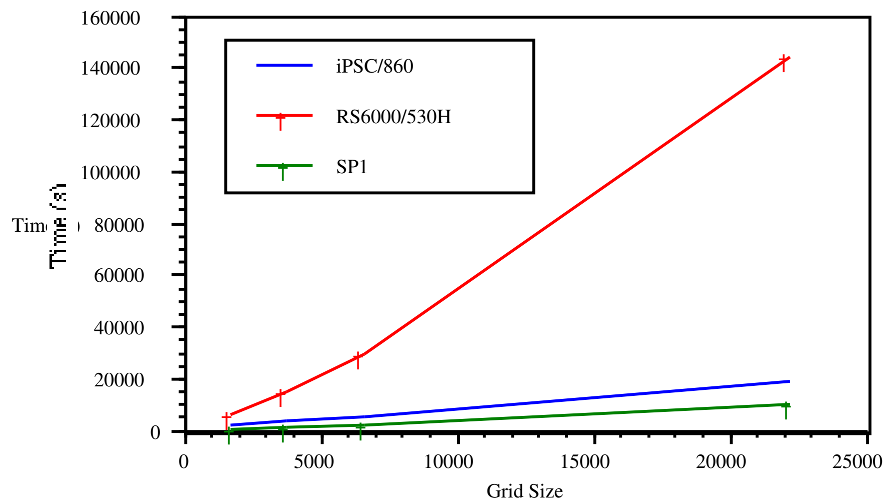

Figure 3.32 FIESTA-HD solution times for increasing grid sizes. (32-processor iPSC/860, RS6000/530H, and 8-processor SP1.)

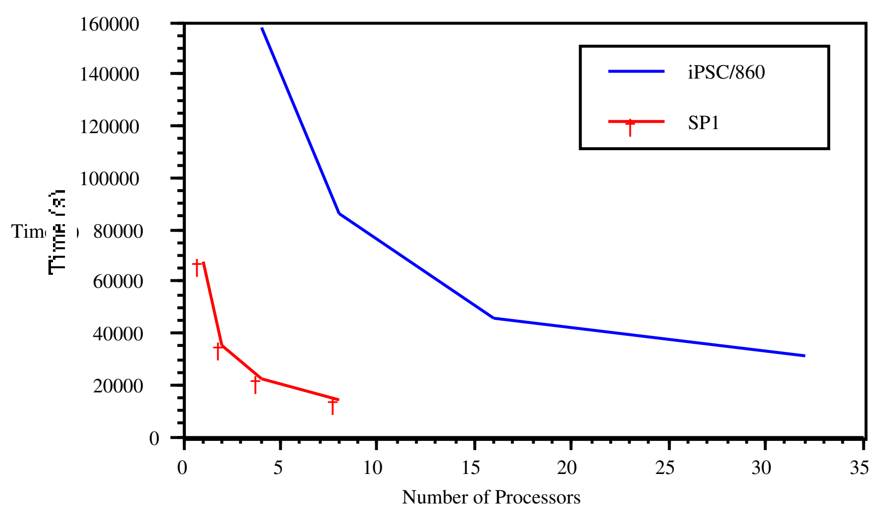

Figure 3.33 FIESTA-HD iPSC/860 and SP1 solution times for n-channel MOSFET.

3.5 Summary

In this chapter, the parallelization of a “dusty deck” semiconductor device solver, PISCES, using the coarse-grain parallelization model has been discussed in-depth. Modifications to the existing code were minimized and localized to insure the integrity of the parallel application. The resulting application has been proven easily portable with versions existing for the Intel iPSC/860, the TMC CM-5, and the IBM SP1/2. Solution capabilities have been greatly enhanced and simulation results have demonstrated good performance for a wide range problems. The parallelization process resulted in the creation of a flexible communication framework as well as an application-area-specific domain decomposition module. These tools were then employed in the coarse-grain parallelization of a second semiconductor device simulator, FIESTA-HD. This application also exhibits excellent parallel performance.