Chapter 4

Application-Oriented Gaussian Elimination in PISCES-MP

The parallel solution of linear systems using Gaussian (direct) elimination has been the subject of much research. However, the necessary algorithmic modifications to integrate parallel direct methods into useful applications are rarely straightforward. The memory limitations, parallel data structures, and partitioning imposed by a parallel application may force extensive algorithmic modifications in the parallel linear solver. This chapter describes the algorithms used by the linear solution step of PISCES-MP and provides a foundation for discussing the adaptive direct/iterative linear solution method presented in Chapter 5.

4.1 Gaussian Elimination and Sparse Matrices

The solution of the matrix problem $\mathbf{A}x = b$ is a key step in the repetitive process used to the solve nonlinear systems in PISCES-MP (and many other PDE applications). Due to the extreme ill-conditioning of the matrices arising from the approximation scheme used by PISCES, Gaussian (direct) elimination is used. The algorithm for Gaussian elimination can be stated very simply as in Figure 4.1. High performance on modern computers can only be achieved by modifying this basic algorithm to efficiently utilize both the memory hierarchy and functional characteristics of modern computer systems [46]. Techniques such as cache blocking, software pipelining, and loop unrolling are often used to improve the performance of matrix algorithms [17].

```
for k = 1 to number_of_equations
    for j = k+1 to number_of_equations
        A(j,k) = - A(j,k) / A(k,k)
    endfor
    for j = k+1 to number_of_equations
        for i = k+1 to number_of_equations
            A(i,j) = A(i,j) - A(i,k) * A(k,j)
        endfor
    endfor
endfor
```

Figure 4.1 Basic Gaussian elimination algorithm.

Another important algorithmic optimization in the implementation of Gaussian elimination is the exploitation of the sparse matrix structure arising from grid-based PDE simulators to limit the amount of memory and floating-point operations required to obtain a solution. In most cases, the number of non-zero entries in a PDE-derived sparse matrix is only a small fraction of the total entries. Accordingly, sparse matrices are rarely stored in a full *n*x*n* array (where *n* is the number of equations). One common storage scheme stores entries in a compressed form by column (shown in Figure 4.2 for a simple 4x4 sparse matrix). For each column, the first array (*column_index*) provides an index into a second array (*row_list*) which contains a list of the non-zero row indices for that column.

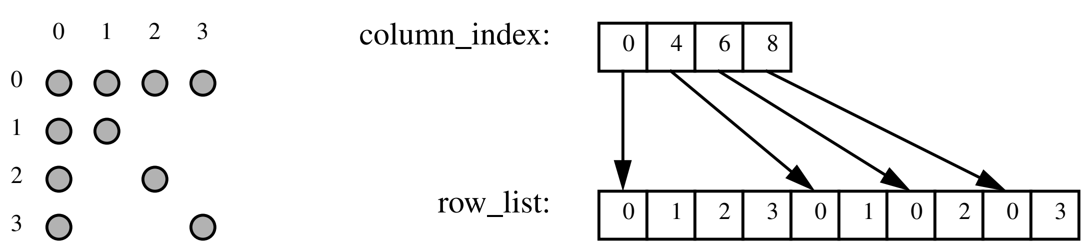

Figure 4.2 Column_index[*i*] points to the beginning of column *i*'s row indices stored in row_list.

During elimination, zero entries in the matrix may become non-zero through the process of fill-in. Minimization of non-zero fill-in by intelligently choosing the order of elimination is crucial in order to limit the amount of computation and memory required. For example, Figure 4.3 shows the resulting fill-in patterns created by two different elimination orderings of the same matrix. In this case, an interchange of the first row and column with the last row and column, respectively, transforms a maximal fill-in into a minimal fill-in. As illustrated, an efficient ordering can greatly reduce both computation time and memory requirements. Obtaining the optimal ordering for a general sparse matrix is an NP-complete problem [18], and research into reordering heuristics is an active field.

The exploitation of sparsity has an additional and unwanted side-effect: the regularity of computation is disrupted. Since efficient computation largely depends upon regularity, many performance optimizations lose their effectiveness. Most sparse methods seek to recapture this performance by imposing regularity upon an irregular sparse structure.

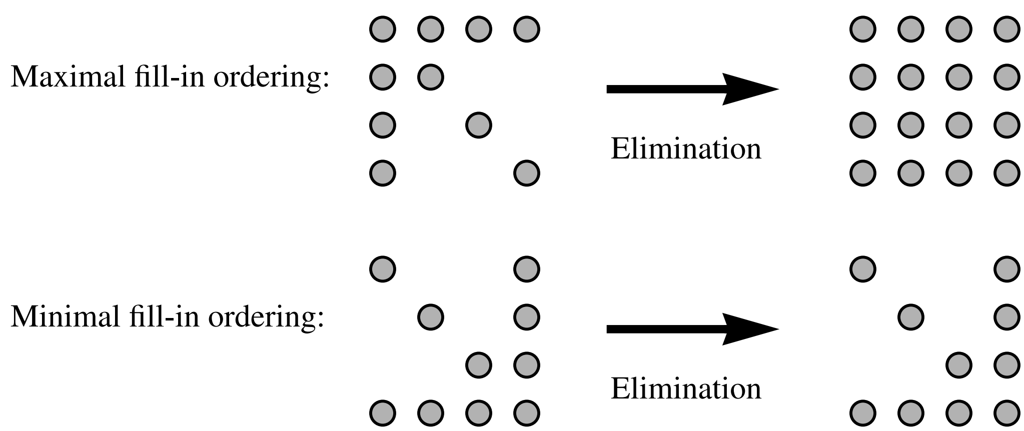

Figure 4.3 Example of the effect of reordering on sparse matrices.

4.2 Serial Multi-Frontal Factorization

The multi-frontal method is one popular method for imposing order upon a sparse system. This approach is an extension of the frontal method first proposed by Irons [31] for finite element applications. The multi-frontal method [10] seeks to take a poorly structured sparse factorization and transform it into a series of smaller dense factorizations. These dense eliminations exhibit a well understood structure and can be made to run well using techniques already developed for dense systems. A thorough understanding of the underlying serial algorithms is crucial for understanding the extensions necessary to adapt this method for use within a parallel application. The following subsections present a brief introduction to the algorithms and data structures used by each of the steps in the PISCES sparse multi-frontal linear solver: reordering and elimination-tree creation, symbolic factorization, numerical factorization, and triangular solution.

4.2.1 Minimum Degree Reordering and Elimination Tree Creation

As previously illustrated in Figure 4.3, the crucial first step in sparse elimination is the computation of a fill-reducing ordering. One widely used ordering algorithm employs the multiple minimum degree (MMD) algorithm [19]. (A thorough discussion of many ordering heuristics including MMD is given in George and Lui [18].) The MMD algorithm uses a local criteria to minimize fill-in. At each stage, the algorithm chooses the equation with the fewest non-zeroes in its column (i.e., the equation with minimum degree) as the next equation to eliminate. After modifying the matrix structure to reflect this elimination, the process is repeated until all equations have been ordered. The standard output of the MMD algorithm (i.e., a simple list of the elimination ordering) is insufficient to represent many of the data independencies inherent in sparse systems. These data independencies can be exploited to improve floating-point performance. Thus, it is often desirable to create a more sophisticated representation of the elimination ordering which captures these data independencies. One common approach uses a tree structure to define a partial ordering of the elimination [36]. This structure is referred to as the *elimination tree*. In an elimination tree, an equation may be eliminated only after all equations below it in the elimination tree have been eliminated. Equations in different branches of the elimination tree have no data dependencies. Figure 4.4 shows the standard MMD ordering and the corresponding elimination tree for the 4x4 matrix of Figure 4.3. The elimination tree captures the fact that equations 1, 2, and 3 must be eliminated before equation 0 but are independent of each other. The traditional MMD ordering cannot represent this independency.

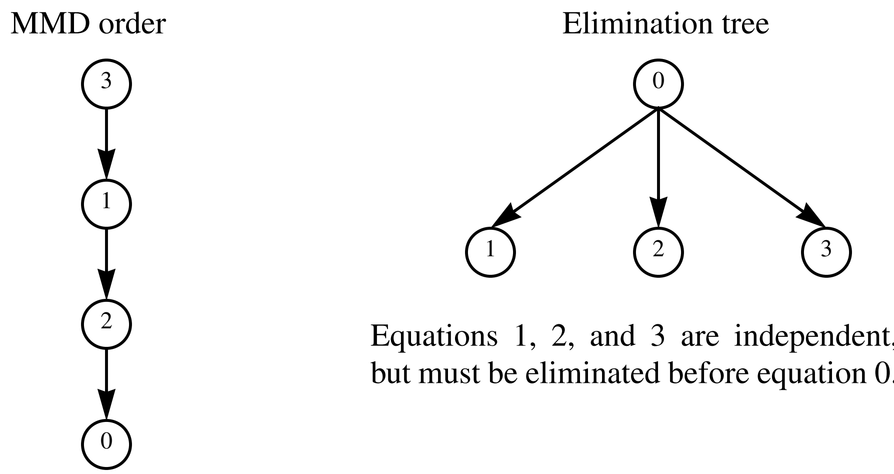

Figure 4.4 Standard MMD ordering and elimination tree representations.

To further improve both symbolic and floating-point performance, the multi-frontal solver used by PISCES exploits the existence of *supernodes* in the sparse matrix structure. A supernode is a grouping of equations with similar non-zero column structure. For example, equations 0 and 1 in Figure 4.5 have identical column structure and may be grouped into a supernode. During processing, the supernode is used to reference the group of equations. Supernodes not only exist in the original matrix structure but also are created by fill-in during elimination. In PDE applications, a large initial set of supernodes usually exists due to the multiple degrees of freedom per mesh point (all of which have identical

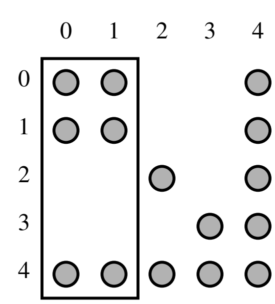

Figure 4.5 Equations 0 and 1 may be grouped into a supernode.

connectivity in the underlying mesh structure). Representing a group of equations with a single supernode reduces the number of data items and the amount computation in the MMD algorithm. Likewise, during numeric factorization, all of the equations comprising a supernode may be eliminated in a single step, requiring fewer gather/scatter operations. In the subsequent discussion of MMD, all equations are placed into supernodes (often size=1) during initialization to simplify data representation.

```
1.  create adjacency-graph structure
2.  compute initial supernodes and compress initial adjacency structure
3.  create a list, sorted by supernode degree, of all nodes
4.  while(supernode list not empty)
5.      foreach (eligible supernode of minimum degree)
6.          remove supernode from degree list, mark it eliminated
7.          mark all immediate neighbors as ineligible in this iteration
8.          add supernode to the elimination tree
9.      endforeach
10.     update degrees of ineligible supernodes, mark them eligible
11.     update list of minimum degrees
12. endwhile
```

Figure 4.6 Reordering and elimination tree creation using the MMD algorithm.

Figure 4.6 presents a high-level description of the MMD algorithm. Line 1 is an initialization step which transforms the application's representation of the matrix into a graph representation suitable for the MMD algorithm. Line 2 examines the adjacency graph for initial supernodes. After creating the initial supernodes, the adjacency structure is updated and references made redundant by the supernodal groupings are removed. Line 3 sorts the supernodes according to their number of neighbors (i.e., the degree of each supernode in the adjacency graph). The resulting list is referred to as the *degree list* and is used by the algorithm to select candidates for elimination. Line 4 controls the ordering process and continues until all supernodes have been eliminated. Line 5 begins the repetitive process of selecting supernodes for elimination. The degree list is examined and the next eligible candidate of minimum degree chosen. If more than one candidate exists, a tie-breaking scheme must be employed [19]. One solution is to choose the lowest-numbered supernode. Once a supernode has been chosen, line 6 performs the first bookkeeping step by removing it from the degree list and marking it as eliminated. Line 7 then marks all supernodes adjacent to the eliminated supernode as ineligible for further consideration during this iteration. (The adjacency graph must first be updated to reflect the changes in its neighborhood to insure consistency in the selection process.) The final task during elimination of a supernode is to place it in the elimination tree (line 8). Placement in the elimination tree is achieved by finding all neighboring supernodes which have been previously eliminated and yet have no parent in the elimination tree. The just-eliminated supernode is then made the parent of these supernodes. The removal of supernodes creates non-zero fill-in, coupling the neighboring supernodes and changing their column structures (i.e., their degrees). Thus, the adjacency structure must be modified to reflect these changes. After all eligible supernodes have been eliminated and added to the elimination tree, line 10 updates the adjacency graph. Once this step updates the degrees of some supernodes, line 11 resorts the degree list in preparation for the next iteration of the algorithm.

The application of the MMD algorithm on a simple finite-volume grid with a single degree of freedom per mesh point is shown in Figure 4.7a-Figure 4.7f. Figure 4.7a shows the initial adjacency structure. (For grid-based PDE applications, the initial adjacency graph is usually identical to the mesh.) In this example, all initial supernodes contain only a single equation. In the first iteration of the algorithm, two candidates for elimination exist (0 and 14) with *degree=2*. The candidates are independent (i.e., they have no connection in common) and may both be eliminated and added to the elimination tree. The adjacency graph is then modified to reflect the elimination. At this stage, no fill-in has been added. The second iteration in the algorithm (Figure 4.7b) presents a large number of candidates with *degree=3*. In this case, supernodes 1, 11, and 12 are chosen and added to the elimination tree. Supernode 1 becomes the parent of supernode 0 as a result of the initial mesh connection between them. Likewise, supernode 11 becomes the parent of supernode 14. With these eliminations, fill-in has been created. One example is the fill-in between supernodes 2 and 3 created by elimination of supernode 1. The updated adjacency graph (Figure 4.7c) reflects the newly created couplings. In the third iteration of the algorithm, supernodes 2 and 13 (*degree=3*) are chosen for elimination. Supernode 2 becomes the parent of supernode 1 in the elimination tree. Due to its earlier connections with supernodes 11 and 12, supernode 13 becomes the parent of both. The fourth iteration is shown in Figure 4.7d. The minimum degree is again 3. Supernode 3 is eliminated and added to the elimination tree as the parent of supernode 2 by way of the link created by earlier fill-in. Also, supernode 10 is eliminated and made the parent of supernode 13. In the fifth iteration (Figure 4.7e), supernode 4 (*degree=3*) is eliminated and added to the elimination tree as the parent of supernode 3. Likewise, supernode 8 is eliminated and made the parent of supernode 10. After updating, the remaining supernodes in the adjacency graph (Figure 4.7f) are fully connected (i.e., they all have identical column structure) and can be grouped into a single supernode. This supernode is then eliminated and the elimination tree is made complete.

Remark: Examining the elimination tree yields valuable insight. The major left and right branches of the elimination tree corresponding to the left and right halves of the underlying mesh structure are data-independent. Their parent supernode separates them just as its corresponding mesh points separate the underlying mesh. This property plays an important role in the future parallelization.

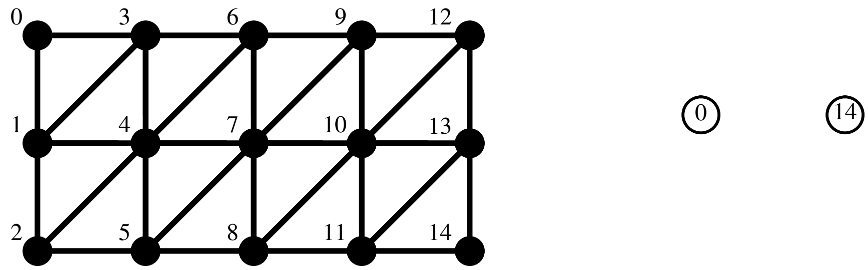

Figure 4.7a Adjacency structure (left) and elimination tree (right) during the first iteration of the MMD algorithm.

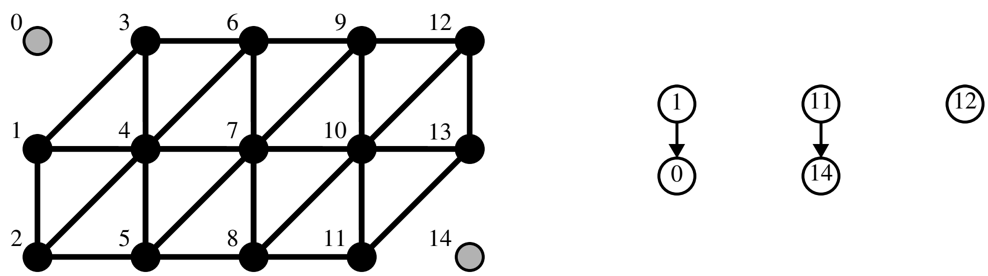

Figure 4.7b Second iteration of the MMD algorithm.

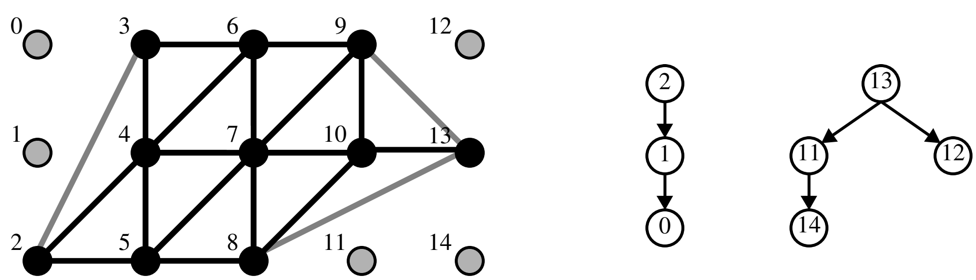

Figure 4.7c Third iteration of the MMD algorithm.

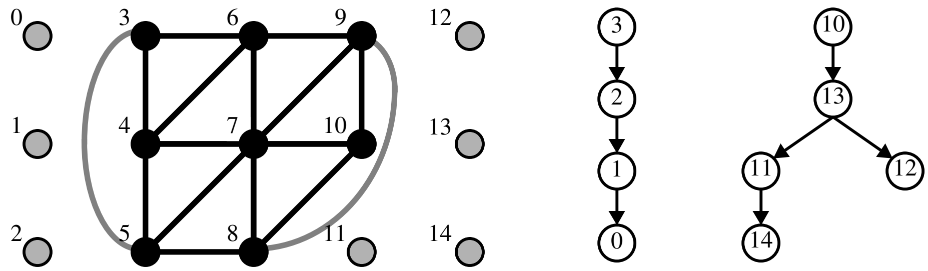

Figure 4.7d Fourth iteration of the MMD algorithm.

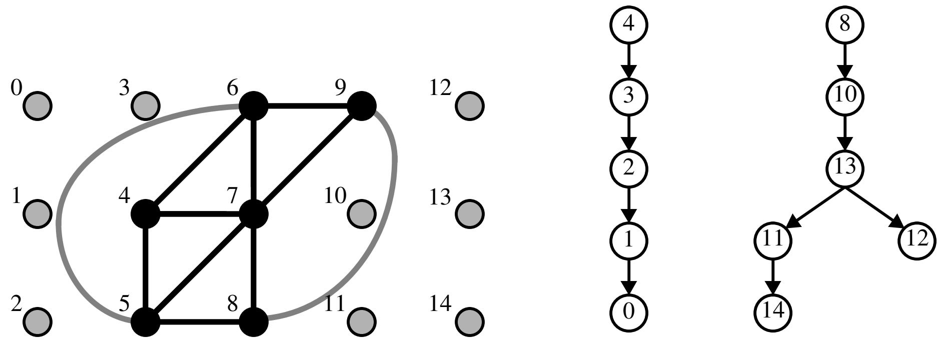

Figure 4.7e Fifth iteration of the MMD algorithm.

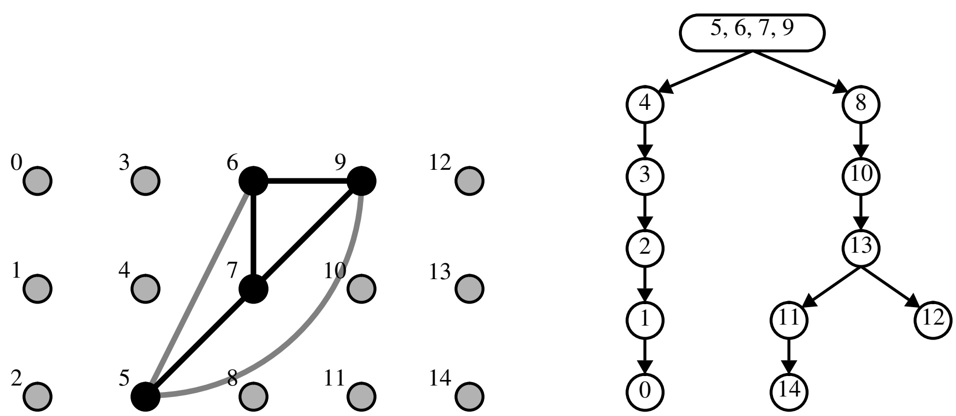

Figure 4.7f Final iteration of the MMD algorithm.

4.2.2 Symbolic Factorization

Once fill-in has been minimized by an equation ordering algorithm such as MMD, a symbolic factorization examines the elimination tree and determines the structure of the filled-in matrix. The profile created by the symbolic factorization is used to create the appropriate data structures for storage and indexing. Non-zero entries in the factorized matrix come from three sources:

- Non-zeroes in the original matrix.
- Non-zeroes created by elimination within the supernodes.
- Non-zeroes created by elimination between supernodes.

The elimination tree and the structure of the original matrix can be used to determine the fill-in from these three sources. First, a depth-first traversal of the elimination tree is made (Figure 4.8). (This ordering insures that no fill-in for a supernode is computed until all of its children have been processed.) Using the depth-first ordering and the original matrix, the three types of fill-in are computed for each supernode. First, the connectivity in the initial matrix structure is added. Next, any additional connectivity created by the formation of the supernode is added. Finally, fill-in is added between the current supernode and its ancestors in the elimination tree caused by the elimination of the children of the current supernode.

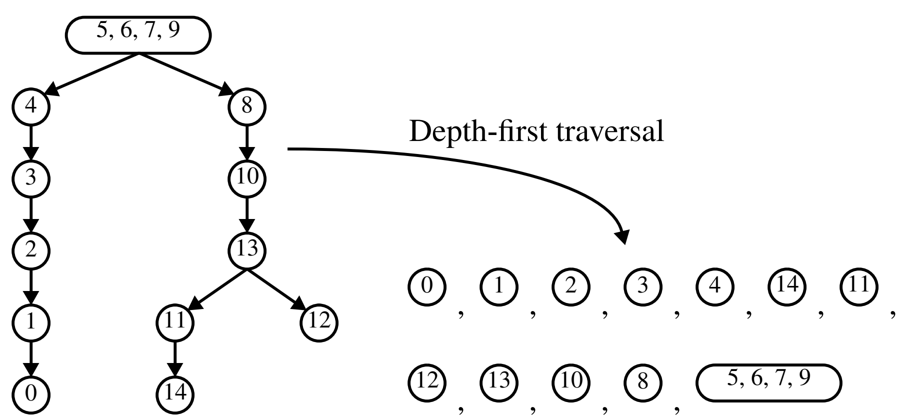

Figure 4.8 Depth-first traversal of the elimination tree used by the symbolic phase.

Once the size and structure of the factored matrix have been computed, the floating point storage must be allocated and appropriate indexing arrays created. Figure 4.9 depicts the storage layout scheme for the factored matrix used by the PISCES multi-frontal solver. This scheme is a variation of the compressed column format presented earlier in this chapter. The matrix is split into three components: the diagonal component (**D**), the column-oriented lower triangle (**L**), and the row-oriented upper triangle (**U**). This data layout is used to improve the efficiency of the triangular solutions.

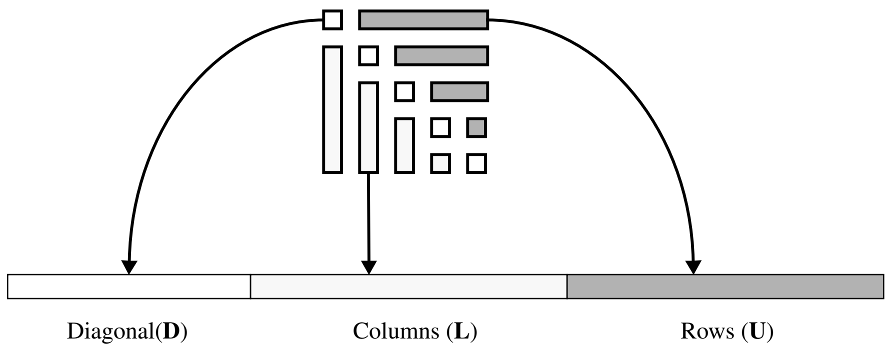

Figure 4.9 Storage layout for the factored matrix.

In addition to the permanent storage needed by the factored matrix, temporary working storage must also be profiled and allocated. Working storage includes space to assemble and factor each dense block of the sparse matrix as well as scratch space for intermediate fill-in values. With appropriate indirect indexing, sections of a single one-dimensional storage structure can be used for both purposes. The size of the array and the indexing can be computed at the same time the matrix factor is profiled.

4.2.3 Matrix Assembly

The next step in the solution is the assembly of the **A** matrix. To accomplish this, the non-zero values must be transferred from the element-block-oriented data structures used in the PDE approximation to the sparse matrix data structure. In a dense matrix, the row and column indices can be used to directly compute the address; however, in an unstructured sparse matrix the row and column numbers must be mapped to an address within the compressed matrix structure. Often, the local equation numbering within each block must first be mapped to the global equation numbering. Next, the global equation numbers are used to look up the address in the compressed structure using the *row_list* and *column_index* (or analogous data structures). In general, this lookup entails searching through the row indices of the selected column to determine the proper offset. Only then may the non-zero values be copied to the sparse structure (Figure 4.10). This procedure is repeated for each non-zero entry and can become expensive. To speed matrix assembly, these addresses may be precomputed, stored in a table, and simply looked up when needed. However, the amount of storage required for these maps can become quite large (roughly the same size as the element non-zero matrices). Thus, memory constraints often limit the amount of optimization which may be performed during this phase.

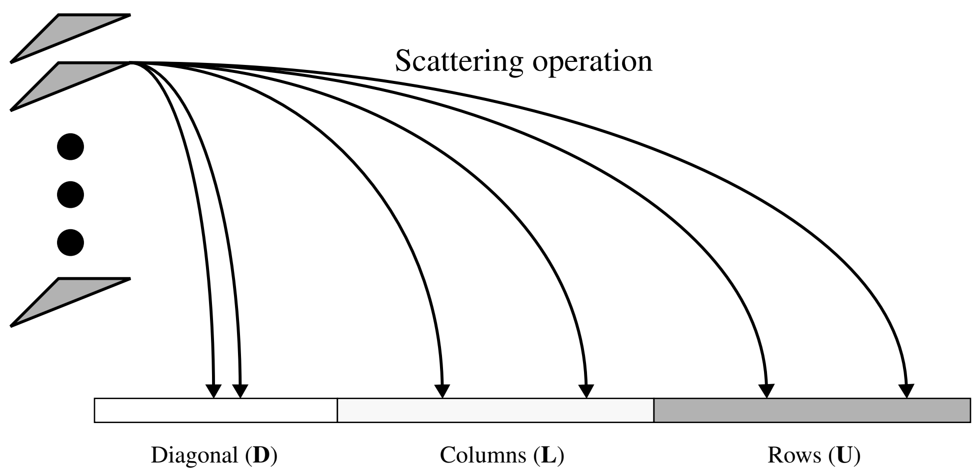

Figure 4.10 Scattering the element (or volume) data structure to the matrix.

4.2.4 Numeric Factorization

The numeric computation may begin once the elimination order has been decided, the data structures allocated, and the initial matrix assembled from the element blocks. The multi-frontal method factors the sparse matrix as a series of smaller dense matrices (also referred to as fronts). Each front is assembled from initial matrix values and fill-in created by earlier eliminations (Figure 4.11). Within a front of size n, a single supernode of size m ($1mn$) is then eliminated. Afterwards, the factored equations are copied to permanent storage and the fill-in is copied into temporary storage for later use (Figure 4.12).

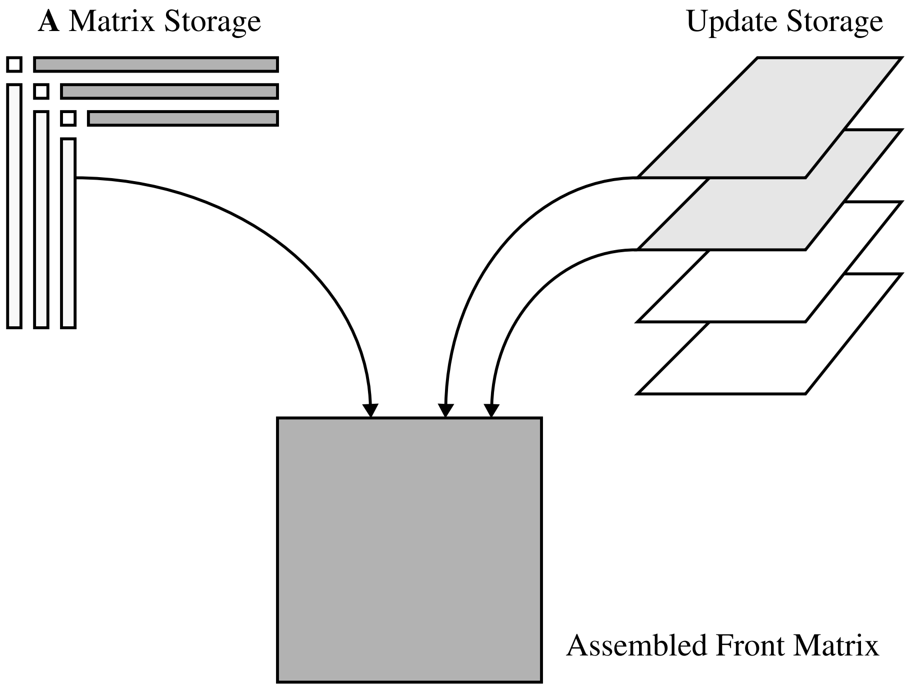

Figure 4.11 Assembly of a front for a supernode with two children.

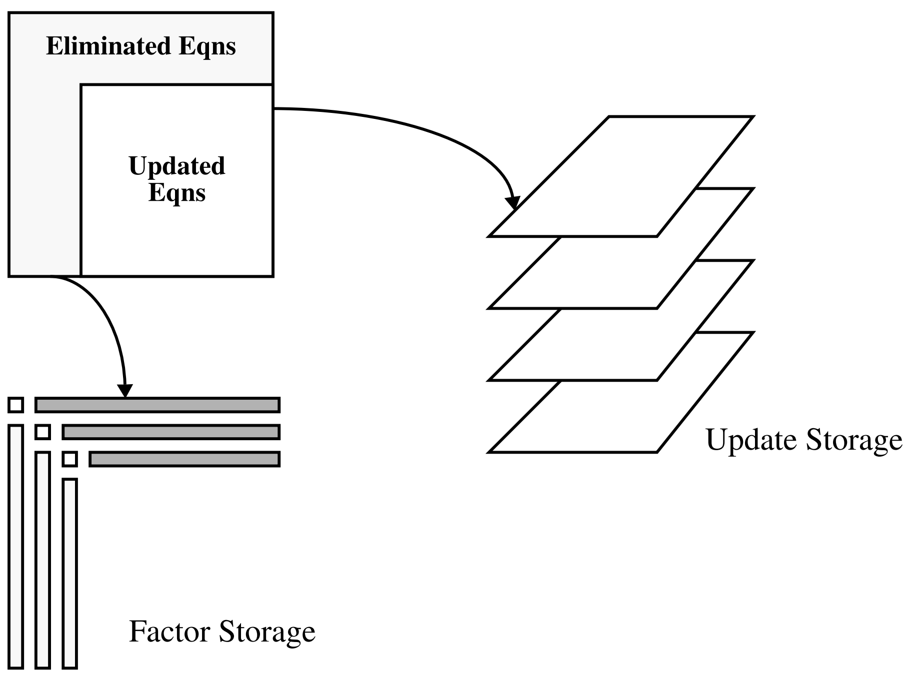

Figure 4.12 Scattering of the factorized front matrix.

```
1.  foreach (supernode in the depth-first traversal of the elimination tree)
2.      create a temporary dense matrix for the supernode's front
3.      copy the supernode's initial values to the front
4.      foreach(child in the elimination tree)
5.          add fill-in to the front created by this child's elimination
6.      endforeach
7.      eliminate the supernode from the front
8.      copy eliminated equations to factor storage
9.      copy fill-in to temporary storage for use by parent
10. endforeach
```

Figure 4.13 Multi-frontal elimination algorithm.

Figure 4.13 describes the multi-frontal factorization algorithm. Line 1 defines the outer loop of the factorization process. The depth-first traversal of the elimination tree used in the symbolic phase to determine the structure of the matrix also determines the order in which the dense blocks are factored. Again, this ordering insures that no supernode is eliminated until all of its children have been eliminated, insuring that all requisite updates to the supernode will be available. Line 2 creates storage for the current (or active) front matrix. Line 3 begins assembly by copying the original matrix values for the supernode into the front matrix's storage. Lines 4 and 5 add the fill-in created by earlier elimination of the current supernode's children in the elimination tree. Line 7 performs the elimination of the supernode within the front. Using well known optimizations for dense matrix operations, this process be made extremely efficient. The dense kernel used in the PISCES solver employs BLAS Level-3 [6] matrix operations which achieve high-performance on many architectures. After elimination, line 8 copies the factorized supernode to permanent factor storage. Line 9 copies the update to the parent in the elimination tree to temporary storage to be used later.

4.2.5 Triangular Solve

The factorization has reduced the matrix into a lower triangular matrix (**L**) and an upper triangular matrix (**U**). Thus, the solution to **A**x=b can be recast into the solution of **LU**x=b. To obtain a solution, first substitute y for **U**x. Then, solving **L**y=b, followed by **U**x=y, leads to a solution. Solving **L**y=b for y is known as forward elimination. The

PISCES solver uses the column-oriented implementation of forward elimination as shown in Figure 4.14. Solving **U**x=y is known as back substitution. The row-oriented implementation used in PISCES is shown in Figure 4.15.

```
for i = 1 to number_of_equations
    y(i) = b(i)
    for j = i+1 to number_of_equations
        b(j) = b(j) - y(i) * l(i,j)
    endfor
endfor
```

Figure 4.14   Column-oriented forward elimination.

```
for i = number_of_equations downto 1
    for j = i+1 to number_of_equations
        y(i) = y(i) - x(j) * u(i,j)
    endfor
    x(i) = y(i) / u(i,i)
endfor
```

Figure 4.15   Row-oriented backward substitution.

4.2.6 Memory Management

When performing Gaussian elimination, memory is often the most limiting resource. Consequently, efficient memory reuse becomes critical. Unfortunately, many general-purpose memory-management tools (e.g., malloc and free) are insufficient to efficiently manage the diverse memory requirements of sparse methods. This often leads to significant memory wastage which limits feasible problem sizes. In the multi-frontal method, each computational phase (reordering, symbolics, etc.) requires both permanent storage and temporary storage. As computation progresses, the amount of permanent memory required increases while the amount of temporary memory decreases (Figure 4.16). Generally, a simple stack-based scheme can be used to allocate memory. The stack is managed by the linear solver, allowing intelligent reuse of memory. Permanent storage is placed at the bottom of the memory stack with temporary values allocated above. At the end of each phase, newly created permanent storage is added and the temporary storage pointer is moved higher in memory for the next phase. This requires that the permanent storage needed in each phase be known beforehand.

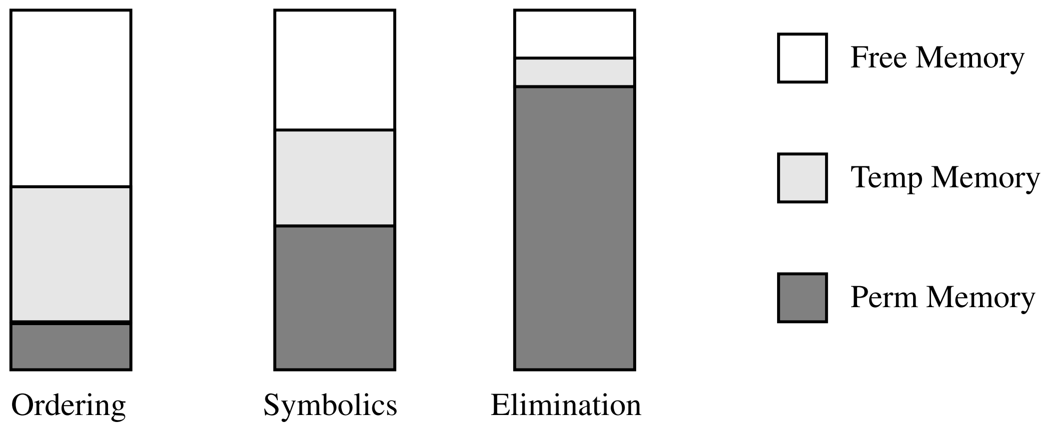

Figure 4.16 Stack-based memory-management scheme.

4.2.7 Serial Results

The statistics for several linear systems arising in the simulations in Chapter 3 are summarized in Table 4.1. The corresponding factorization, and triangular solution timings on an IBM RS6000/590 workstation are given in Table 4.2. To maintain an application-oriented perspective, it is important to note the relative importance of the direct solver within the application. For this reason, the matrix formation time is also listed in Table 4.2. The matrix formation time includes the time taken by PISCES to create the initial matrix as well as the time taken to map these non-zero values into the sparse matrix data structure (i.e., assemble the matrix). The results show that both the formation and solution of the matrix contribute significantly to the overall application runtime. The triangular solution step requires roughly one-tenth as much execution time compared to the factorization. These results show that the multi-frontal method is suitable for solving the linear systems arising in PISCES.

Table 4.1 Matrix statistics for PISCES-MP simulations.

	Equations	Initial Non-Zeroes	Factor Size (words)	Operations (Mflops)
2K CMOS Inverter	4,821	101,562	346,977	16
8K CMOS Inverter	23,532	508,182	2,605,716	234
11K CMOS Inverter	32,184	697,572	3,786,660	382
18K CMOS Inverter	55,509	1,210,308	8,010,117	1,172
6K Bipolar Transistor	17,793	387,660	2,403,045	288
6K Photo-Emitter	16,800	254,810	1,279,214	37

Table 4.2 IBM RS6000/590 execution times.

	Form Time	Factor Time	Solve Time
2K CMOS Inverter	0.80s	0.29s	0.04s
8K CMOS Inverter	4.46s	2.67s	0.26s
11K CMOS Inverter	6.23s	4.20s	0.37s
18K CMOS Inverter	11.36s	11.08s	0.76s
6K Bipolar Transistor	3.63s	2.92s	0.23s
6K Photo-Emitter	2.87s	0.45s	0.04s

4.3 Application-Oriented Parallel Factorization

The use of independently developed direct linear solvers within serial applications is commonplace. A well defined interface can make the integration of a linear solver into a serial application straightforward. Despite a well defined interface, there exists an underlying close coupling between the application and the linear solver. This coupling arises from the shared perspective provided by the existence of a shared memory space.

For example, matrix assembly (the mapping from the application's element storage to the linear solver's matrix storage) can be easily accomplished by exploiting this common perspective. Popular serial libraries conform to standard interfaces while relying upon the underlying common view provided by the shared memory. However, a coarse-grain parallel application distributes the creation of the matrix among multiple processors and multiple memory spaces. Without the existence of a shared memory view, it is unlikely that a common perspective can be maintained. Clearly, this data distribution adds considerable complexity to the interface between the linear solver and the application.

A parallel application divides the simulation mesh and seeks to equitably distribute computation in the PDE approximation and physical model evaluation. In addition, the application decomposition must respect any underlying physical constraints in the simulation domain. (In simulation domains with large numbers of materials, complex models, or diverse element types, the decomposition can become quite complicated.) These considerations preclude imposing a linear solver's decomposition upon the application. Thus, the creation of sparse direct libraries which can be easily integrated into parallel applications becomes more problematic. Three potential approaches to integrate a parallel linear solver into an already distributed application are summarized below:

- After the creation of the distributed matrix **A**, gather the pieces into a single processor's memory to form a fully assembled **A** matrix. This matrix can then be given to the parallel sparse matrix solver using an existing serial interface. This approach preserves the existing interface by reconstructing a common perspective. Internally, the linear solver partitions and redistributes the matrix to optimize its computation. One primary drawback to this approach lies in the overhead incurred in the gathering and scattering of the matrix and the subsequent gathering and scattering of the results. This gather/scatter time may prove substantial and correspondingly reduce parallel efficiency. Also, the solution of large problems may be infeasible if the matrix is too large to be fully assembled.
- Redistribute the matrix for the linear solution step without creating a fully assembled copy. After the matrix is formed (according to the application decomposition), it is redistributed to match solver decomposition. This approach

forces the application to maintain two decompositions, one for the matrix creation step and one for linear solver, increasing the application complexity. Also, the overhead of redistributing the matrix negatively impacts efficiency. On the other hand, both the application and the linear solver are allowed to employ appropriate decompositions.

- Modify the linear solver to accept the application's data distribution. In this case, a single decomposition is maintained throughout all phases of computation and no explicit data redistribution is required. Management of a single decomposition simplifies the application; however, the application decomposition may be less appropriate for the linear solver and reduce its efficiency.

For the development of PISCES-MP, the first approach was deemed undesirable due to the inherent problem-size limitations. The third approach was chosen over the second approach for three primary reasons: a single decomposition reduces program complexity, no explicit data reshuffling was required, and the application partitioning did not appear to significantly hamper the linear solver. It remains unclear whether the second approach would prove viable. A full study of such a scheme remains an area for future study.

Considerable effort has been channelled into the development of parallel multi-frontal linear solvers and has produced excellent results as first reported by Lucas [35]. Further improvements have made the parallel multi-frontal method suitable for solving the large-scale problems produced in PISCES-MP. Both Gupta [26] and Rothberg [46] present detailed discussions. However, most relevant work continues to assume a common perspective between the application and the linear solver. The importance of application integration is often overlooked and trade-offs are rarely considered. Consequently, most of the work assumes that the linear solver maintains its preferred decomposition.

Based in the initial work of Lucas, a parallel multi-frontal direct solver which works within a distributed application has been developed for PISCES-MP. This linear solver uses the application's mesh decomposition to partition the corresponding matrix. The domain decomposition algorithm used by PISCES-MP creates mesh separators in a top-down fashion as described in Section 3.2.2.7. These separators partition the resulting matrix and lead to a *global* elimination tree as shown in Figure 4.17. Each processor factors its local portion of the matrix independently and then cooperates with other processors to factor the shared portions of the matrix. The following subsections present the algorithmic changes required to create an application-oriented direct solver and the modifications to each of the major phases in the linear solution (reordering, symbolics, etc.) will be discussed. The resulting solver will be shown to provide excellent solution capabilities despite its unorthodox decomposition.

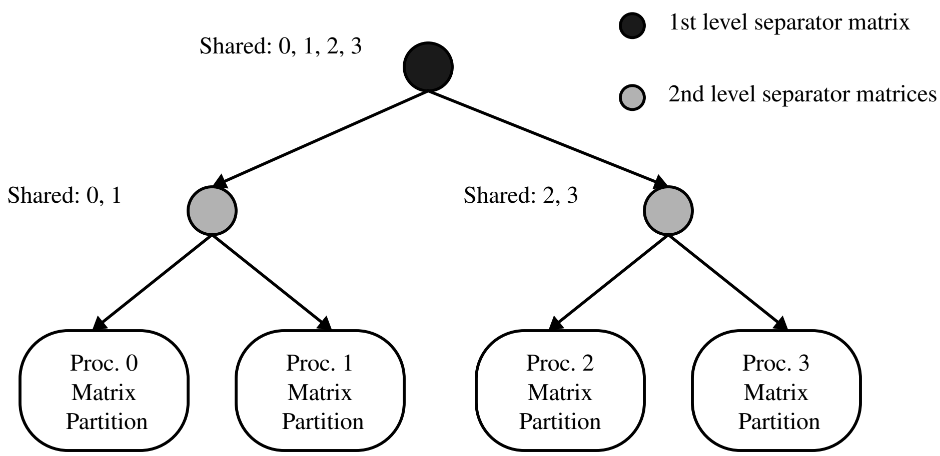

Figure 4.17 Global elimination tree based on a 4-processor partitioning.

4.3.1 Two-Phase Equation Ordering with MMD

The application's mesh decomposition imposes an ordering upon the shared equations and defines the higher levels of the elimination tree (Figure 4.17). However, each processor continues to order its local equations to minimize local fill-in. Despite the lack of a global perspective, each processor contains all of the necessary information to compute a local ordering. The MMD algorithm is modified to order only local equations while accounting for the connectivity and fill-in added by the shared equations. A local elimination tree is created and then the shared supernodes are added at the top of this elimination tree. The modified MMD algorithm is shown in Figure 4.18. This algorithm differs little from the standard MMD algorithm (Figure 4.6). During initialization, Line 4 searches the adjacency graph and marks all supernodes containing shared equations as ineligible for selection by MMD. However, these supernodes are accounted for in the MMD algorithm in order to obtain an accurate snapshot of connectivity within the matrix. By making them ineligible, the elimination of the shared supernodes is delayed until all local supernodes have been eliminated and added to the elimination tree. Once the local elimination tree is computed, line 14 is responsible for adding the shared supernodes at the top of the elimination tree. The supernodes are added to the elimination tree according to their level in the decomposition (i.e., the first level separator is at the top, the second level separator directly below the first, etc.).

```
1.  create adjacency-graph structure
2.  compute initial supernodes and compress initial adjacency structure
3.  create a list, sorted by supernode degree, of all nodes
4.  mark all shared supernodes as ineligible
5.  while(eligible supernode list not empty)
6.      foreach (eligible supernode of minimum degree)
7.          remove supernode from degree list, mark it eliminated
8.          mark all immediate neighbors as ineligible in this iteration
9.          add supernode to the elimination tree
10.     endforeach
11.     update degrees of ineligible supernodes, mark them eligible
12.     update list of minimum degrees
13. endwhile
14. add the separator supernodes to the top of the e-tree
```

Figure 4.18 Reordering and elimination tree creation using the MMD algorithm.

To illustrate the distributed MMD ordering, Figure 4.19 shows a two-processor partitioning of the example grid introduced in Figure 4.7a. Equations 6-8 comprise the separator and are duplicated on both processors. The partition for processor 0 contains equations 0-8 and the partition for processor 1 contains equations 6-14. In the underlying mesh, processor 0 is ordering the left half of the adjacency graph while processor 1 is ordering the right half of the adjacency graph. Each processor orders its local equations according to the modified MMD algorithm and constructs a local elimination tree. After initialization, only equations 0-5 and equations 9-14 are eligible in processor 0 and processor 1, respectively. During the first four iterations of the modified MMD algorithm, the equations chosen by the two processors are identical to the equations chosen by the original MMD algorithm (shown in Figure 4.7a-Figure 4.7d). This first differences arise in the fifth iteration (Figure 4.20a). On processor 0, supernode 4 is chosen and added to the elimination tree. This choice is consistent with the serial MMD algorithm. However, processor 1 has only one eligible candidate remaining, supernode 9. In the serial case, supernode 8 was chosen during this iteration, but it resides in the separator and is ineligible for elimination. Processor 0 eliminates its final eligible supernode and adds it to the elimination tree (Figure 4.20b). Once all local supernodes have been eliminated, the MMD algorithm halts (Figure 4.20c). Finally, each processor adds the shared supernode to the top of its elimination tree.

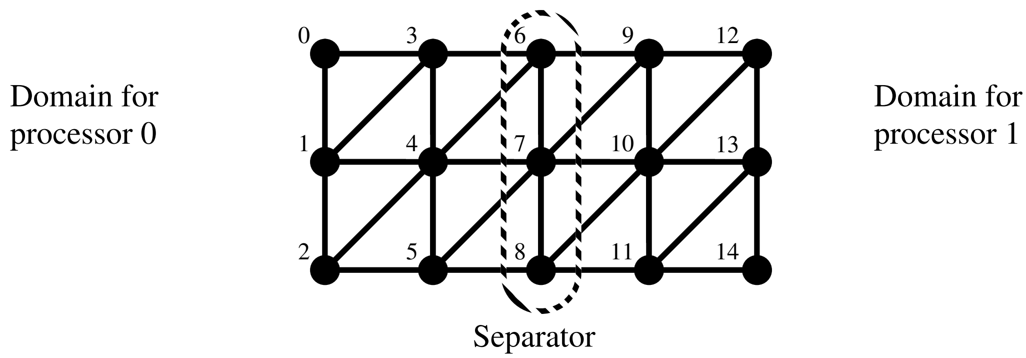

Figure 4.19 Two processor grid decomposition of a simple grid.

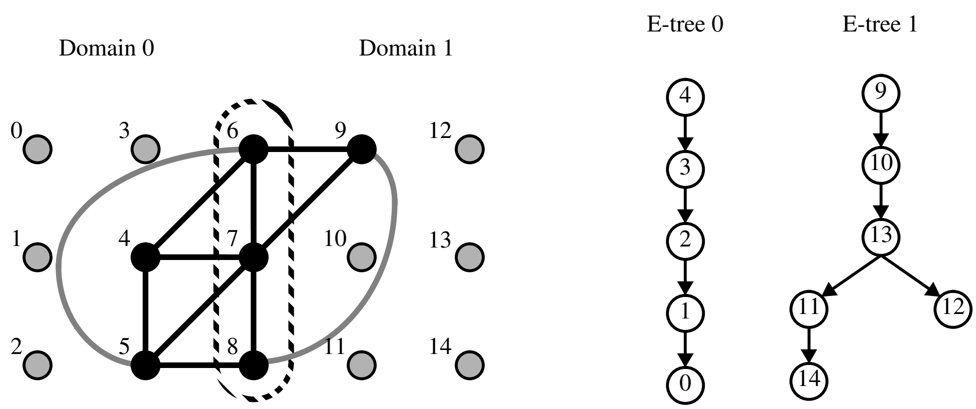

Figure 4.20a Fifth iteration of the modified MMD algorithm.

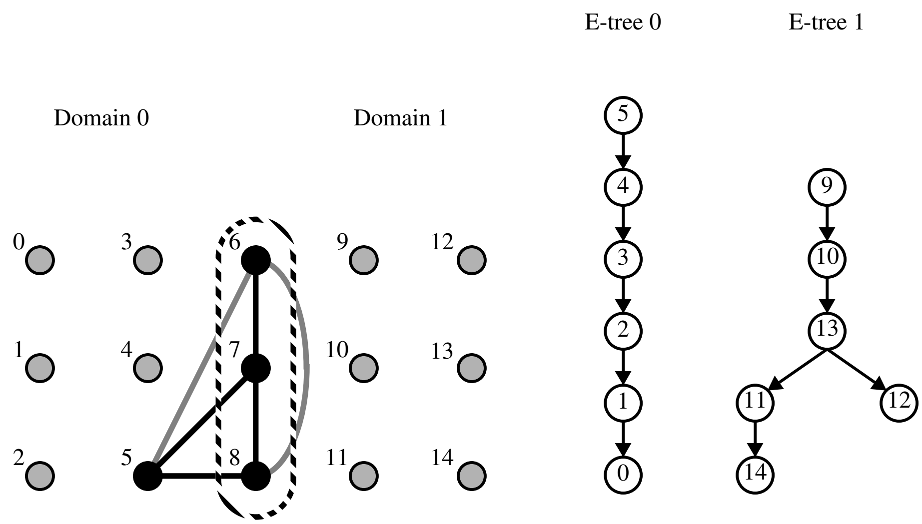

Figure 4.20b Sixth iteration of the modified MMD algorithm.

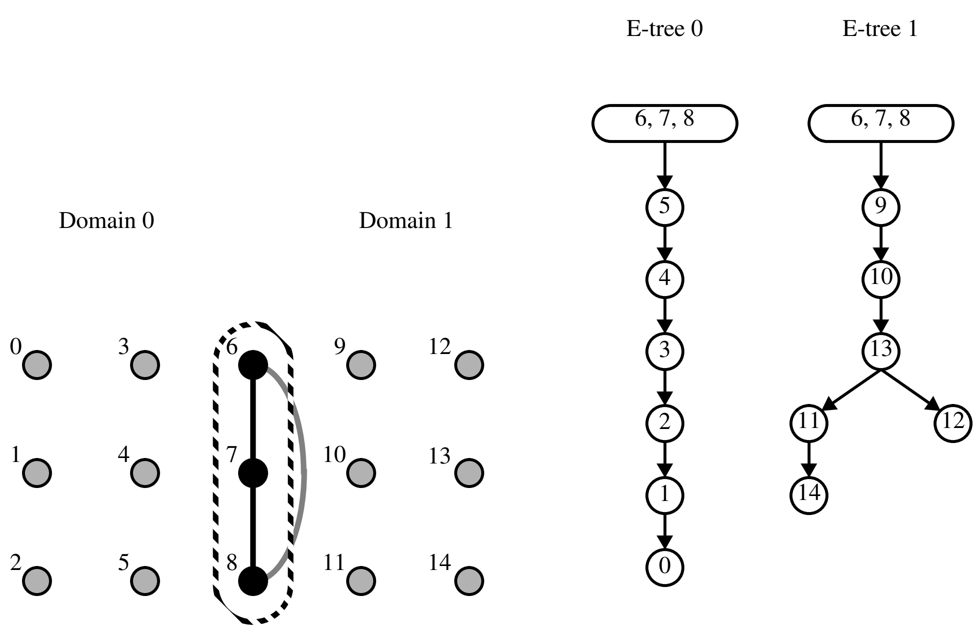

Figure 4.20c Addition of shared supernode to the elimination tree.

4.3.2 Distributed Symbolics

After ordering the equations locally and creating a local elimination tree, each processor performs a symbolic factorization to determine the non-zero structure of its locally factored matrix. Non-zero entries in each processor's locally factored matrix are created in four ways:

- Non-zeroes in the original local matrix.
- Non-zeroes created locally by elimination within the supernodes.
- Non-zeroes created locally by elimination between supernodes.
- Non-zeroes created non-locally between shared supernodes.

The first three processes are the same as in the serial case and can be computed using the same approach as described in Section 4.2.2. The fourth mechanism arises from the distributed nature of the computation. To illustrate, Figure 4.21 shows the fill-in created after local elimination for a simple grid with a four-processor decomposition. The elimination of the local domains creates fill-in between equations in shared supernodes. However, no processor can independently construct a correct picture of the fill-in between the separators. For example, neither processor 0 nor processor 1 has sufficient local information to completely determine fill-in between their second-level and their first-separators. Processor 0 is aware of the fill-in between the second-level separator (equations 1 and 4) and equation 6 in the first-level separator but is ignorant on the fill-in between the second-level separator and equation 8 in the first-level separator. The reverse is true for processor 1. (The same relationship exists between processor 2 and processor 3.) If the processors are to successfully cooperate in the factorization of the shared supernodes, they all must possess an accurate representation of the matrix structure. To correct any inconsistencies, the processors communicate and exchange their respective contributions to the shared supernode fill-in after computing the local fill-in. Storage and indexing arrays are then allocated based upon this complete view of the shared supernodes.

In addition to creating the sparse matrix data structures discussed in Section 4.2.2, the parallel symbolic phase also sets up temporary buffers and processor-to-shared-column mappings for the upcoming factorization and triangular solution steps.

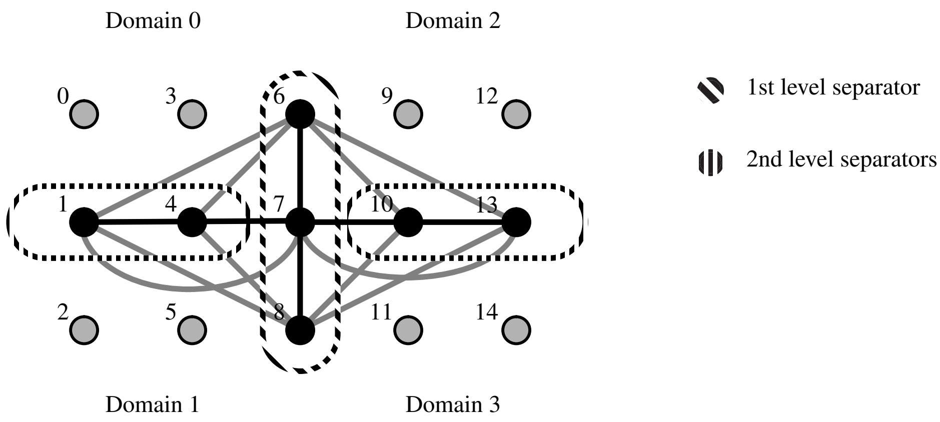

Figure 4.21 Shared supernode fill-in for a four-processor decomposition.

4.3.3 Matrix Assembly

In parallel, the mapping of the matrix from the distributed application's data structures to the distributed sparse matrix data structure (Figure 4.22) requires no communication and can be performed independently by each processor. This independence is a consequence of both employing a single decomposition throughout the application and the existence of separate processor memory spaces in the coarse-grain programming model. Partial values corresponding the shared equations exist on each processor and will be globalized along with the fill-in during the factorization step.

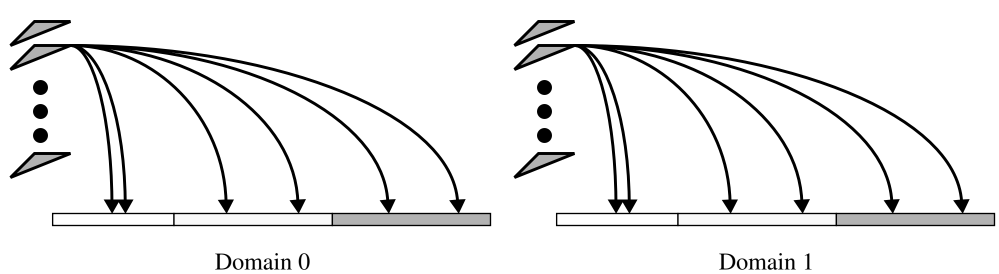

Figure 4.22 Two-processors scattering the distributed element (or volume) data structure to the distributed matrix.

4.3.4 Numeric Factorization

The overall structure of the multi-frontal algorithm remains intact when modified for parallel execution. Factorization of the local supernodes proceeds as in the serial solver; however, modifications must be made in order to factor the shared supernodes. Figure 4.23 shows the modified factorization algorithm. Lines 7-11 are added to the serial algorithm (Figure 4.13). These lines test each supernode for locality. If the supernode is local, it is factored using the existing dense matrix kernels. If the supernode is shared a parallel dense kernel is employed.

```
1.  foreach (supernode in the depth-first traversal of the elimination tree)
2.      create a temporary dense matrix for the supernode's front
3.      copy the supernode's initial values to the front
4.      foreach(child in the elimination tree)
5.          add fill-in to the front created by this child's elimination
6.      endforeach
7.      if(supernode is local)
8.          eliminate the supernode from the front using serial kernel
9.      else
10.         eliminate the supernode from the front using parallel kernel
11.     endif
12.     copy eliminated equations to factor storage
13.     copy fill-in to temporary storage for use by parent
14. endforeach
```

Figure 4.23 Parallel multi-frontal elimination algorithm.

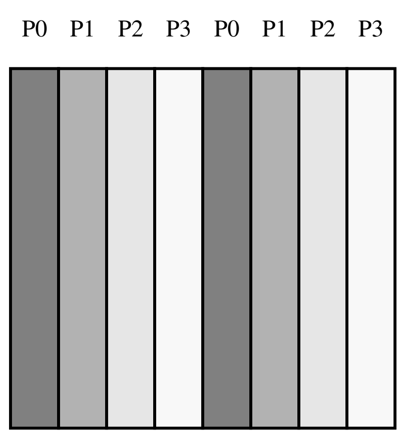

Figure 4.24 Four-processor distribution for the parallel dense factorization. Each column block contains of four equations.

```
1.  exchange columns
2.  foreach(column_block)
3.      if(owner = me)
4.          invert(pivot_block)
5.          send(pivot_block)
6.      else
7.          receive(pivot_block)
8.      endif
9.      foreach(column_block that I own)
10.         update(column_block by pivot_block)
11.     endforeach
12. endforeach
```

Figure 4.25 Parallel column-block dense matrix elimination.

The factorization of the shared supernodes employs a variant of the parallel column-block algorithm described in Rothberg's thesis [46]. This approach decomposes the dense matrix into a collection of blocks. Each block consists of one or more columns. (In PISCES-MP, column blocks normally contain four columns.) Column blocks are distributed among the processors in a round-robin fashion as shown in Figure 4.24. The processors cooperate to factorize the dense matrix employing the algorithm summarized in Figure 4.25. Once the column blocks have been distributed among the processors (line 1), each processor loops over the column blocks comprising the matrix (line 2). Line 3 tests the ownership of the current column block. The owning processor executes lines 4 and 5 to invert the column block and send the results to the other processors. The processors which do not own the column block wait to receive the inverted column block (line 7). Each processor then updates its column blocks with the pivot block in lines 9-11.

The column block approach employs a one-dimensional decomposition of the dense matrix. Two-dimensional matrix decompositions are generally more scalable and exhibit better efficiency [8]. The benefits of a two-dimensional decomposition are greatest for small problem sizes or large numbers of processors. Nevertheless, this one-dimensional decomposition provides good performance on problems arising in PISCES-MP. Rather than pursue improvements in the factorization of the shared supernodes, an adaptive direct/iterative scheme has been developed and will be presented in Chapter 5.

4.3.5 Triangular Solve

Like the factorization, the triangular solutions maintain the basic structure of the serial algorithms with appropriate modifications to account for the shared supernodes. The column-oriented forward elimination exhibits a “fan-in” structure with respect to the elimination tree. Each processor performs the forward elimination on its local supernodes and then coordinates with other processors to perform the forward elimination on the shared supernodes. In this fashion, information flow upward through the elimination tree and “fans in” to the shared supernodes. On the other hand, the row-oriented back substitution exhibits a fan-out structure. First, the processors cooperate to compute a solution for the shared supernodes which the “fans out” from the top of the elimination tree and is used by each processor to compute the solution for its local supernodes.

```
foreach i = 1 to number_of_equations
    y(i) = b(i)
    if(shared(i)) then parallelsum(y(i))
    if(owner(i) = me) then
        foreach j = i+1 to number_of_equations
            b(j) = b(j) - y(i) * l(i,j)
        endforeach
    endif
endforeach
```

Figure 4.26 Parallel column-oriented forward elimination.

```
foreach i = number_of_equations downto 1
    if(owner(i) = me) then
        foreach j = i+1 to number_of_equations
            y(i) = y(i) - x(j) * u(i,j)
        endforeach
        x(i) = y(i) / u(i,i)
        send(x(i))
    else
        receive(x(i))
    endif
endforeach
```

Figure 4.27 Parallel row-oriented backward substitution.

Figure 4.26 presents the parallel forward elimination algorithm. The parallel algorithm functions identically to the serial algorithm as the local supernodes are processed. When a shared supernode is encountered, line 3 synchronizes the processors and accumulates the partial results into a consistent value. Line 4 allows only the owner of this column to perform the updates to other shared b_j's. (By default, each processor is the owner of its local supernodes.)

Figure 4.27 presents the parallel backward substitution algorithm. The body of the outer loop has been modified to test for shared supernodes. When a shared supernode is encountered in line 2, the owner is allowed to compute the solution for the shared supernode (lines 3-6) and subsequently send the result to the other processors (line 7). Processors which do not own a shared supernode simply await the result at line 9. Once the shared supernodes have been processed, computation "fans out" to the local domains and computation proceeds as in the serial algorithm.

4.3.6 Parallel Results

The execution times required for several of the matrices arising in the simulations presented in Chapter 3 have been studied to characterize the performance of the parallel multi-frontal solver within PISCES-MP. These matrices provide insight into the issues surrounding application-oriented linear solution methods. Results obtained on an IBM SP2 with RS6000/590 processors are presented in this section. Results obtained on the Intel iPSC/860, the TMC CM-5E, and the IBM SP1 are provided in Appendix B. Likewise, a detailed summary of all execution times discussed in this section are in Appendix B. Statistics for the smaller linear systems have already been summarized in Table 4.1. Statistics for the larger matrices are summarized here in Table 4.3.

The first problems demonstrate the feasibility of the parallel method. However, the high performance of the SP2 and the small subdomains generally make the use of more than 8 or 16 processors unwarranted. Figure 4.29 shows the execution times for the matrix solution within the bipolar transistor simulation. The serial factorization time is 2.9s seconds. With 8 or more processors, factorization time is well below 1 second. The linear solver achieves speedups of 1.9x on 2 processors, 2.6x on 4 processors, 3.7x on 8 processors, 4.4x on 16 processors, and 4.9x on 32 processors. Given the high performance of the RS6000/590 processor, using four to eight processors provides the bulk of the achievable benefits for such a small problem. The execution times for the matrices arising from the simulation of the smallest CMOS inverter grid are shown in Figure 4.30. A single processor can factor the matrix in 0.29 seconds. Speedups of 1.9x on 2 processors, 2.6x on 4 processors, and 3.2x on 8 processors are obtained. On 8 processors, the matrix is factorized in 0.09s. Again, the diminishing sizes of the subdomains coupled with the speed of the RS6000/590 make the addition of more than 8 processors unnecessary. The execution times for the matrices arising from the simulation of the 11k CMOS inverter grid are shown in Figure 4.31. Speedups of 1.9x on 2 processors, 2.4x on 4 processors, 4.5x on 8 processors, 5.8x on 16 processors, and 7.2x on 32 processors are obtained. The factorization time has been reduced from 2.67s to 0.59s using 8 processors. The larger sizes of the subdomains mean that 16 and 32 processors continue to marginally improve the factorization time. The results from 11k CMOS inverter (Figure 4.32) exhibit behavior roughly equivalent to the 8k CMOS results. The results from the matrices arising from the simulation of the largest CMOS inverter grid (Figure 4.33) are better. Speedups of 1.9x on 2 processors, 3.1x on 4 processors, 4.8x on 8 processors, 6.6x on 16 processors, and 8x on 32 processors are obtained. Using 16 processors, solution time has been reduced from 11.1s to 1.7s.

Table 4.3 Matrix statistics for large PISCES-MP simulations.

	Equations	Initial Non-Zeroes	Factor Size (words)	Operations (Mflops)
56K Photo-Emitter	168,000	3,248,211	35,327,976	435
112K Photo-Emitter	336,000	6,808,035	72,065,231	2,211
512K Photo-Emitter	1,536,000	32,186,675	392,841,030	72,238

The performance characteristics of the factorization in the bipolar and CMOS inverter simulations are determined by many factors. The simulation grids are relatively small and unstructured. In most cases, there is insufficient work to warrant using a large number of high-performance processors. Also, when the factorization only takes a fraction of a second, the wisdom of adding more processors must be questioned. In these unstructured grids, the matrix partitioning imposed by the application creates a substantial load imbalance in the factorization. For example, Figure 4.28 shows the number of equations in the higher-level separators of the 18k CMOS inverter grid. At the second level, one separator is almost twice as large as the other. At the third level, separator sizes vary by up to 30%. Notice that the left-hand branch of the tree has separators sized 204, 294, and 294 equations while the right-hand branch has separators sized 204, 177, and 210 equations. The simulations of these unstructured grids lead to large load imbalances affecting the linear solver. Nevertheless, overall application execution time has been improved substantially (as discussed in Chapter 3). If the linear solver becomes a bottleneck in future simulations, the addition of a two-decomposition approach to free the linear solver of the application's decomposition might become an attractive area for future work.

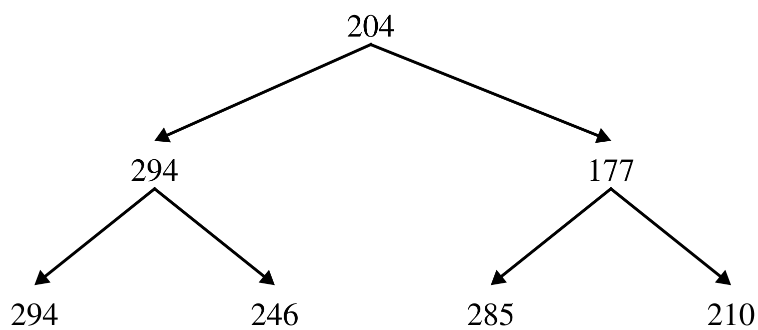

Figure 4.28 Number of equations in the top-level separators for the 18k CMOS grid simulation.

The second set of matrices studied arise from the photo-emitter simulations presented in Chapter 3. The factorization times for the smallest grid are shown in Figure 4.34. This small grid exhibits behavior similar to the bipolar and small inverter grids. Execution time is reduced from 0.45s to 0.07s using 16 processors (a speedup of 6.4x). Once again, such a small problem contains sufficient work for only a small number of high-performance processors. The 56,000 grid point (168,000 equation) simulation shown in Figure 4.35 is significantly larger than previously discussed problems. This simulation was too large to run on 1 or 2 SP2 processors. Using the four-processor result as a baseline, speedups of 2x on 8 processors, 4x on 16 processors, 7.5x on 32 processors, and 13x on 64 processors are obtained. Figure 4.36 shows the factorization times for the 112,000 grid point simulation. Eight processors was the smallest configuration capable of this simulation. Using the eight-processor result as a baseline, speedups of 2x on 16 processors, 3.8x on 32 processors, 7.1x on 64 processors, and 11.6x on 128 processors are obtained. Finally, Figure 4.37 shows the factorization times for the 512,000 grid point simulation. Sixteen processors was the smallest configuration capable of this simulation. Using the sixteen-processor result as a baseline, speedups of 2x on 32 processors, 3.6x on 64 processors, and 5.2x on 128 processors are obtained. This nonsymmetric matrix consisted of 1,536,00 equations and was solved in under 8s using 128 processors. The performance of the PISCES-MP solver on these large-scale matrices demonstrate the suitability of direct methods for the solution matrices arising in large-scale simulations.

Figure 4.38 and Figure 4.39 present speedup with an application-oriented perspective. Within PISCES-MP, the matrix factorization is only one component of the simulation. Matrix formation and assembly is an equally important task for which the simulation grid partitioning has been partitioned. Generally, linear solution performance is not indicative of overall application performance. Figure 4.38 shows the factorization speedup, the formation and assembly speedup, and their combined speedup for the 18k CMOS inverter simulation. Ultimately, the combined speedup is of the most interest. Despite the fact that the partitioning creates a load imbalance in the linear solver as discussed above, combined performance is much better. For example, 16 processors achieve a 6.6x speedup in the factorization while the formation and assembly achieves a 13.5x speedup on 16 processors. Using the combined execution times, nearly an order-of-magnitude improvement in execution time is realized. Figure 4.39 shows the factorization speedup, the formation and assembly speedup, and their combined speedup for the 112k photo-emitter simulation. In this example, the computations are relatively well balanced. Using the 16-processor results as a baseline, the factorization speedup is11.6x and the formation and assembly speedup is 10.9x on 128 processors. Using their combined execution times, their overall speedup is 11.4x on 128 processors.

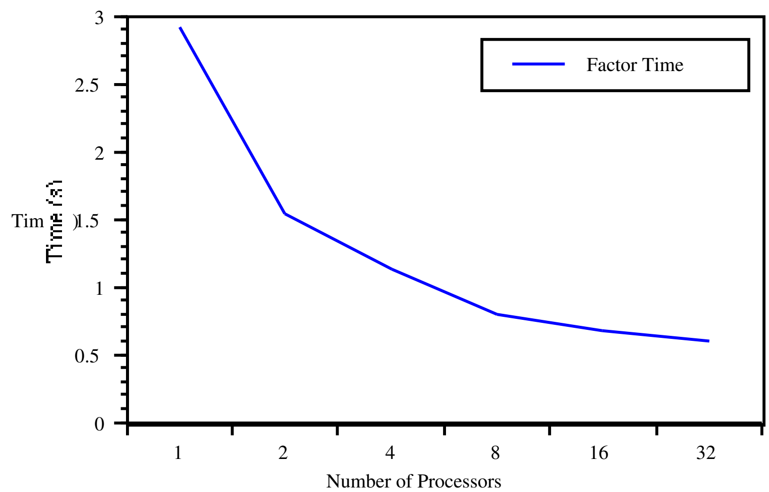

Figure 4.29 SP2 factorization times for the bipolar transistor simulation.

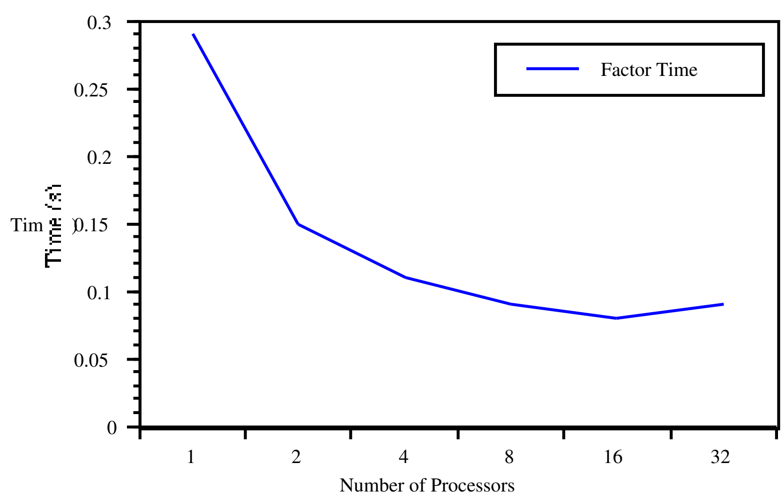

Figure 4.30 SP2 factorization times for the 2k inverter simulation.

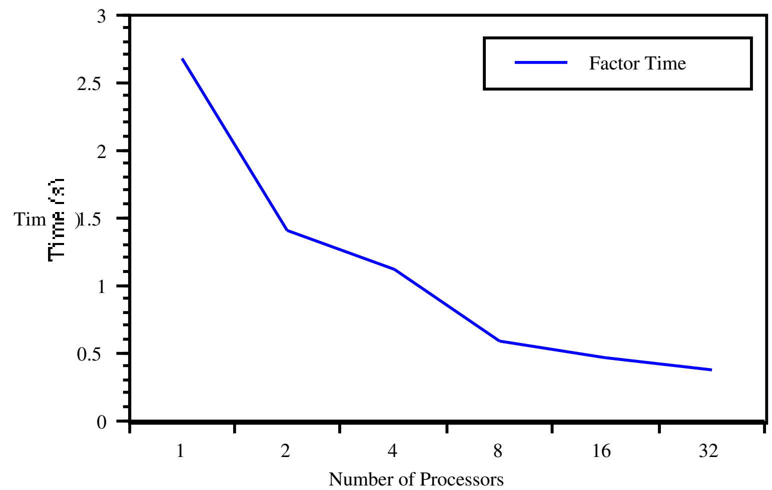

Figure 4.31 SP2 factorization times for the 8k inverter simulation.

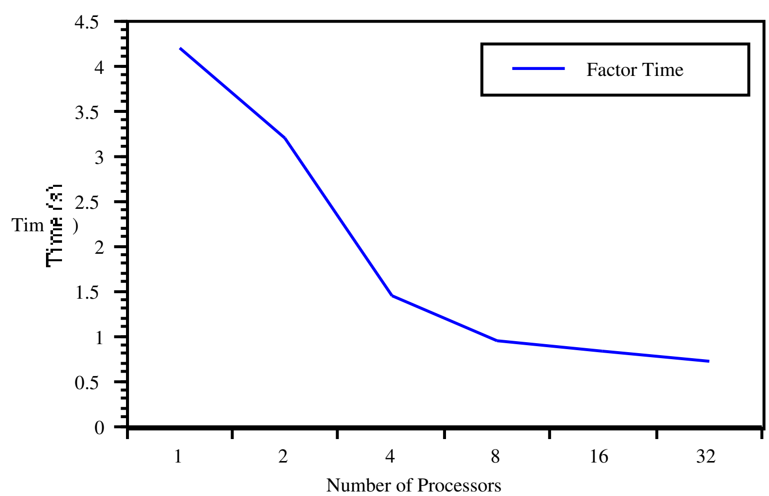

Figure 4.32 SP2 factorization times for the 11k inverter simulation.

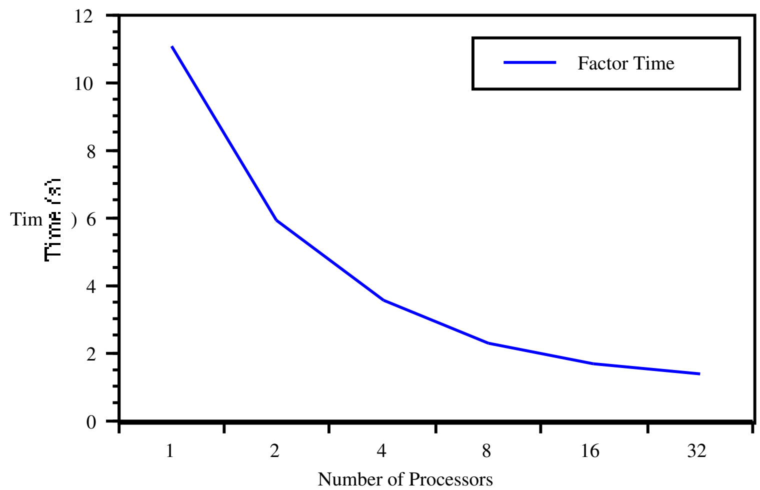

Figure 4.33 SP2 factorization times for the 18k inverter simulation.

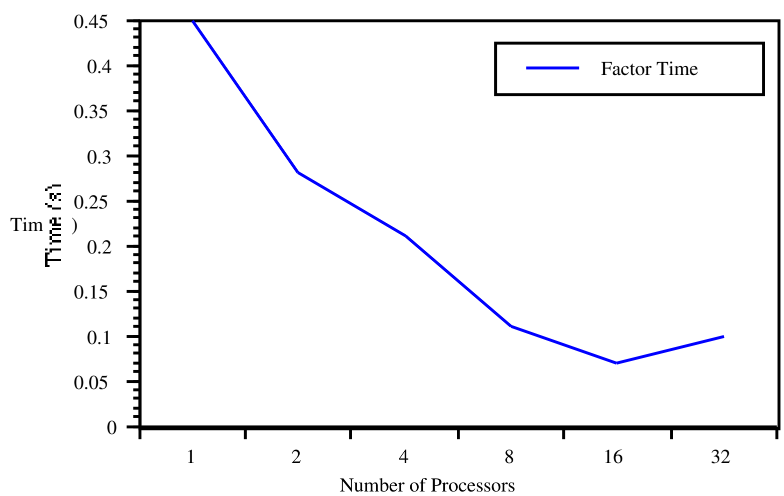

Figure 4.34 SP2 factorization times for the 6k photo-emitter simulation.

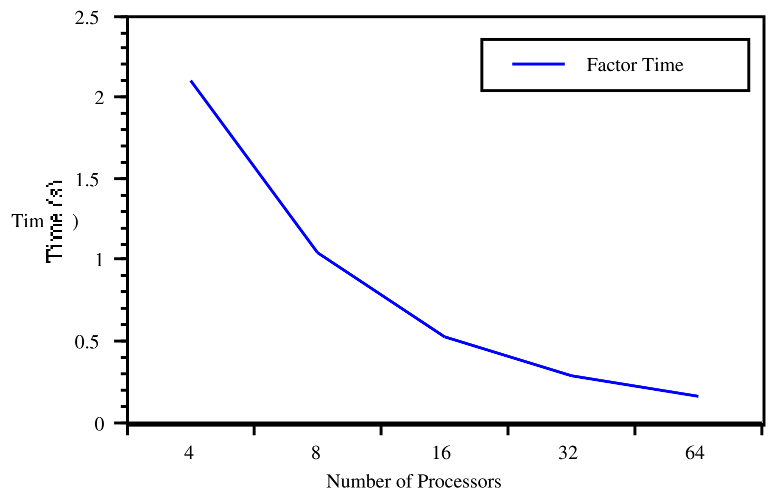

Figure 4.35 SP2 factorization times for the 56k photo-emitter simulation.

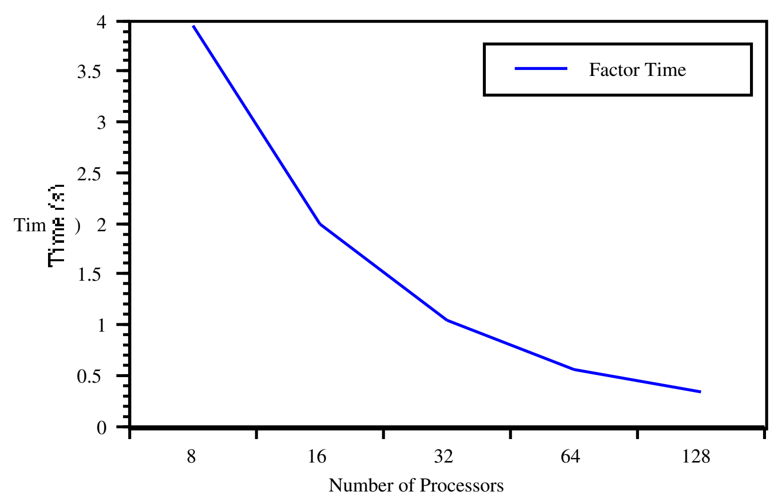

Figure 4.36 SP2 factorization times for the 112k photo-emitter simulation.

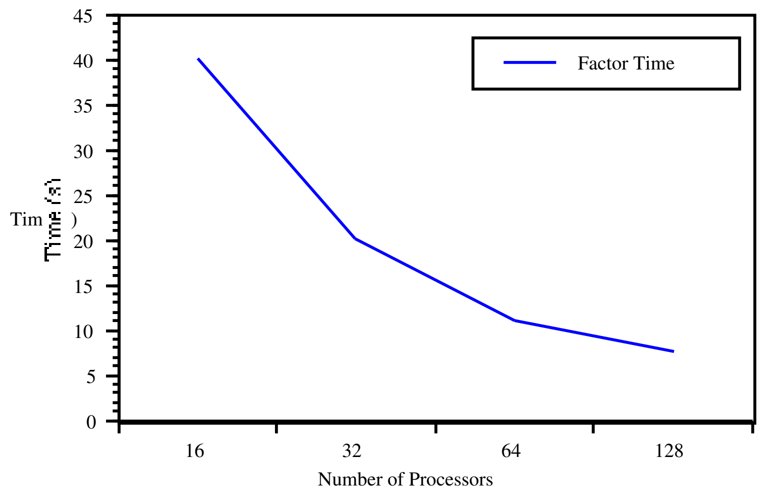

Figure 4.37 SP2 factorization times for the 512k photo-emitter simulation.

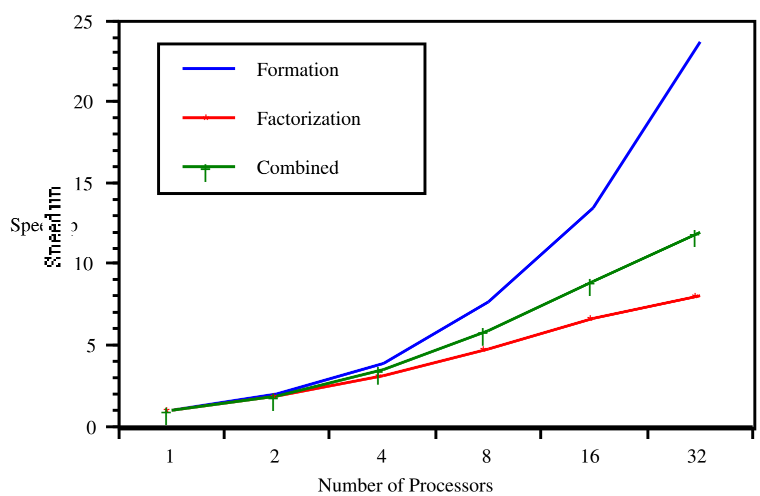

Figure 4.38 Individual and combined speedups for 18k inverter simulation.

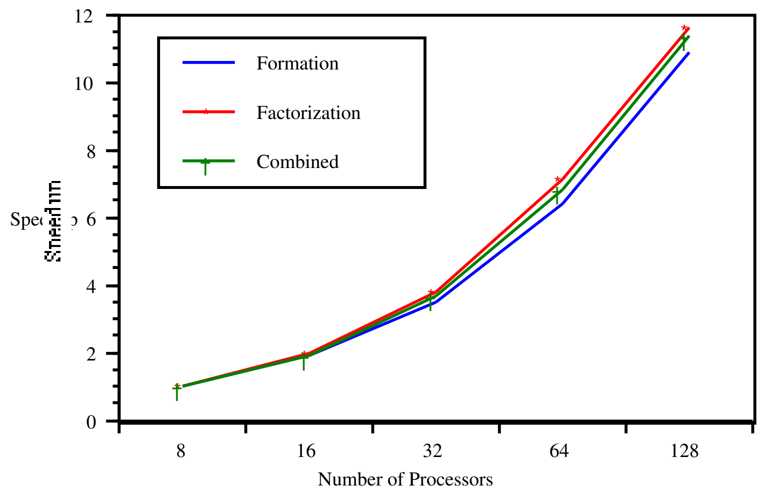

Figure 4.39 Individual and combined speedups for the 112k photo-emitter simulation.

4.4 Summary

In this chapter, the feasibility of using parallel direct methods within a coarse-grain parallel application has been demonstrated. The parallel multi-frontal method described provides acceptable performance for both small and large PISCES-MP simulations. In the smaller simulations, matrix factorization times have been reduced to fractions of a second using eight to sixteen processors. For larger numbers of processors, the high performance of the SP2, small subdomain sizes, and application-imposed load imbalances inhibit further improvements in execution times. However, viewed from an application perspective, the linear solver is sufficient to allow good overall application performance. For larger problems, the linear solver provides excellent performance. Good scalability has been demonstrated on up to 128 processors for problems of up to 1.5 million equations. These results show the ability to efficiently utilize both medium-scale and large-scale parallel computers. Moderately parallel computers (8-16 processors) are suited for moderately sized problems and can provide substantially improved execution times for these simulations. Typically, highly parallel computers (64-128 processors) are intended to solve "grand-challenge" problems. Results have demonstrated that these machines provide excellent solution capabilities when appropriately utilized. Improvements to the methods presented in this chapter can only improve these solution capabilities.