Chapter 5

An Adaptive Linear Solution Method for PISCES-MP

This chapter presents an adaptive direct/iterative linear solution method developed for PISCES-MP. This method uses the elimination tree created by the direct solver to characterize the data flow within the sparse system. Using this information, a partial factorization of the matrix is performed. The remaining equations are then solved using an iterative method which has been grafted onto the direct method. This approach is capable of reducing the memory requirements within the linear solver as well as improving the parallel scalability.

5.1 Solution Strategy

The solution of linear systems is one of the most resource intensive tasks within many grid-based PDE solvers, including PISCES-MP. The application-oriented parallel direct method used in PISCES-MP and presented in Chapter 4 provides solution capabilities which are excellent for large problems and adequate for moderately sized problems. Nevertheless, several strategies for improving these capabilities were also outlined in Chapter 4. Some of these approaches, such as two-dimensional partitioning of the shared supernodes, have great potential for improving the linear solver. However, none of these improvements address more fundamental shortcomings inherent in direct methods. For instance, large memory requirements are burdensome in both serial and parallel environments while the high volume of data traffic required in a parallel environment inhibits parallel efficiency. Iterative linear equation solvers which use less memory and require less communication can be employed to overcome these liabilities. Unfortunately, iterative methods are generally less robust and exhibit slow convergence on ill-conditioned matrices such as those arising in PISCES-MP. To illustrate the trade-offs, a PN diode was simulated by PISCES-MP on a single processor using both direct and iterative linear solvers. The memory requirements and execution times are shown in Table 5.1. Although the direct solver required fives times more memory than the iterative solver, it obtained a solution almost six times faster than the iterative method. After an thorough investigation of iterative techniques in PISCES, Rafferty [45] concluded that direct methods were far more stable and much faster than iterative approaches.

Table 5.1 PN diode serial linear solution times: direct vs. iterative.

	Solution Time	Memory (kwords)
Iterative	88.55s	2,006
Direct	15.00s	9,391

Many schemes have been proposed to take advantage of the complementary natures of direct and iterative methods. For instance, the use of direct methods to generate incomplete factorizations subsequently used as preconditioners in iterative methods is commonplace [7]. This approach seeks to improve convergence without reducing the size of the matrix (i.e., no equations are completely eliminated from the system). An alternative approach is used in the substructuring of structural mechanics problems. In these methods, the substructures can be solved independently using either direct or iterative methods. A consistent treatment of their connecting interfaces insures a correct global solution. A complete discussion of traditional substructuring as well and its applicability to parallel computing using the Finite Element Tearing and Interconnecting (FETI) technique is given by Farhat and Roux [14]. An approach similar to substructuring can be found in the solution of fluid-structure-interaction problems as discussed in Shakib's thesis [48]. In this case, direct solvers are employed against the ill-conditioned structural subdomains while iterative methods are used to solve the fluid subdomains.

A similar scheme was applied by Lyzenga, et.al. [39] to create a hybrid linear method for the parallel solution of finite element problems. In this hybrid approach, each processor uses direct elimination on its local equations while the shared equations are solved using a parallel conjugate gradient algorithm [11]. Figure 5.1 illustrates the breakdown of iterative and direct equations for a four-processor decomposition. As in both the substructuring and the fluid-structure-interaction approaches, the linear solvers were strictly compartmentalized and separated by rigid domain boundaries. This lack of flexibility presents two primary drawbacks. First, the use of direct elimination to solve an entire subdomain can create large dense matrix blocks associated with the boundaries. The resulting memory consumption may limit problem sizes. Secondly, in a parallel environment, as more processors are added for a fixed problem size, the number of the iterative equations grows while the number of direct equations diminishes. Consequently, as the number of processors grows, poor convergence behavior may result as the burden on the iterative method increases.

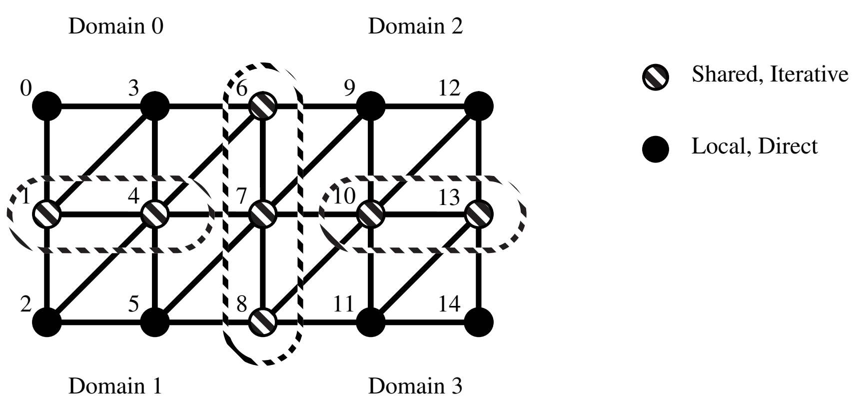

Figure 5.1 Four-subdomain hybrid decomposition.

Despite the success of the substructure-based and hybrid solution methods, the constraints imposed by the maintenance of rigid domain boundaries must be relaxed in order to fully exploit the complementary aspects of direct and iterative methods. An approach allowing arbitrary boundaries between the direct and the iterative subdomains would allow more flexibility. For example, Figure 5.2 shows a flexible allocation of the direct and iterative equations which does not respect the boundaries of the four-processor partitioning (unlike the example in Figure 5.1). In this case, two processors (0 and 1) employ a direct solver on some of their local supernodes while using an iterative solver on the remaining local equations as well as all separator equations. The other two processors (2 and 3) only use the iterative solver on the first-level separator equations while employing the direct solver for both the local equations and the second-level separator equations.

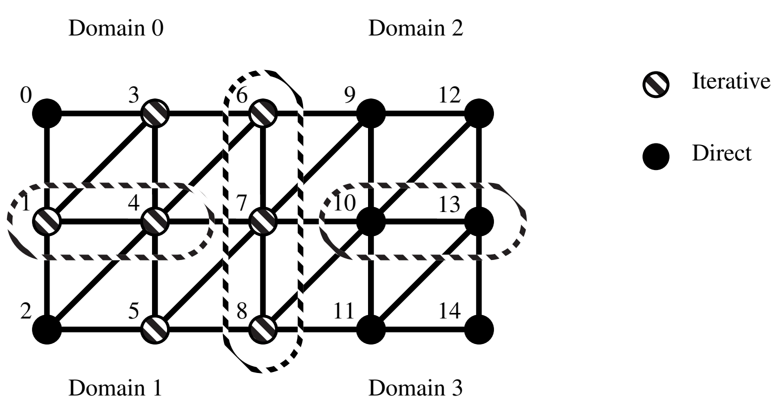

Figure 5.2 Flexible decomposition for four subdomains.

Both machine-dependant and problem-dependant criteria can be used to determine the proper balance between the direct and the iterative equations. The list of potential criteria includes the following:

- Memory limitations which determine the amount of direct elimination (and fill-in) possible.
- The number of equations which can be efficiently solved by the iterative method.
- The boundaries created by a parallel problem partitioning.
- The boundaries created by a problem partitioning based on the physical constraints in the problem domain.

In addition, a combination of these criteria may be applied. These criteria assume that the linear solution method is completely general and allows any combination of direct and iterative solution (including fully direct and fully iterative). Also, the iterative method should be de-coupled from the direct method and easily replaceable to allow a wide range of iterative techniques to be employed.

The multi-frontal solver presented in Chapter 4 provides a robust framework for the construction of the adaptive direct/iterative linear solver described above. First, the notion of supernode-based (i.e., front-based) partial elimination can be easily adapted for the creation of a reduced linear system to be solved with an iterative method. At each step in a multi-frontal factorization, a supernode is eliminated from the system and updates to the remaining system are created. Normally, these updates are used in later supernodal eliminations; however, these updates can also be used to form a reduced matrix problem to be solved using an iterative method. Secondly, the elimination-tree representation of the data flow during elimination can be analyzed to determine the proper balance of equations between the direct and iterative solvers. The elimination-tree representation is sufficiently general to include simple elimination trees produced by standard ordering algorithms as well as more complex elimination trees resulting from a parallel partitioning or a decomposition demanded by the underlying physics. Once the elimination tree is created, the linear solver can analyze it and construct the partitioning for the direct and iterative solvers. Both internal factors such as memory consumption and external factors such as physical constraints in the problem can be considered during this analysis.

5.2 Serial Algorithm

Figure 5.3 shows the high-level algorithm employed by the adaptive direct/iterative solver. Line 1 creates an elimination tree using a method appropriate for the problem domain and hardware environment. Line 2 analyzes the resulting elimination tree and determines the balance between the equations to directly elimination and the equations to be solved iteratively. Line 3 the computes the both the matrix structure for factored equations and the structure of the reduced iterative system. In line 4, the matrix is transferred from the application's data structures to the linear solver's data structures. Line 5 performs a partial factorization to create the reduced system. Then, line 6 creates a reduced right-hand-side vector for the reduced linear system by performing a forward elimination using the factored equations. Line 7 solves the reduced system using an appropriate iterative algorithm. Finally, line 8 computes a solution to the eliminated equations by backward substitution using the solution to the reduced system.

1. create elimination tree
2. analyze the elimination tree to determine balance of direct and iterative equations
3. compute structure of factor and reduced system
4. load matrix
5. partial factor
6. reduce residual
7. solve reduced system using iterative method
8. recover interior

Figure 5.3 High-level direct/iterative solution algorithm.

The algorithm described in Figure 5.3 is applicable to a wide range of PDE-based problems. Moreover, this approach has benefits for both serial and parallel architectures. The following sections discuss the major modifications and additions necessary to create a serial implementation of this adaptive direct/iterative approach based upon the multi-frontal solver presented in Chapter 4. The subsequent parallelization of this method will be presented in Section 5.3.

5.2.1 Symbolic Factorization

Traditionally, the symbolic factorization determines the structure of the matrix and allocates appropriate indexing and storage arrays. For the adaptive direct/iterative approach, the symbolic factorization first analyzes the elimination tree to determine which equations are to be directly eliminated and which equations are to be solved iteratively. Subsequently, the partial elimination must be performed symbolically in order to obtain the connectivity of the reduced system. Finally, the traditional functions of a symbolic factorization are performed to profile and allocate the data structures required by the linear solver.

The bottom-up flow of data through the elimination tree presents the only constraint on the direct/iterative partitioning. This constraint is satisfied by insuring that once a supernode has been declared iterative, all of its ancestors in the elimination tree must also be declared iterative. To achieve this behavior, a bottom-up ordering of the elimination tree is used in the analysis. As shown in Figure 5.4, a bottom-up ordering can be computed by sorting the supernodes according to their depth in the elimination tree.

Remark: When assembling a front and eliminating a supernode during numeric factorization, the multi-frontal method relies upon the assumption that all children of the current supernode have also been eliminated and have created updates to the current supernode. Marking all ancestors of an iterative supernode as iterative prevent this assumption from being violated.

Although the symbolic factorization has been designed to accept an arbitrary elimination tree, the following discussions and examples will be based upon an elimination tree produced by the MMD algorithm as described in Section 4.2.1. Elimination trees based upon other ordering algorithms or created to observe physical constraints can be processed in an analogous fashion.

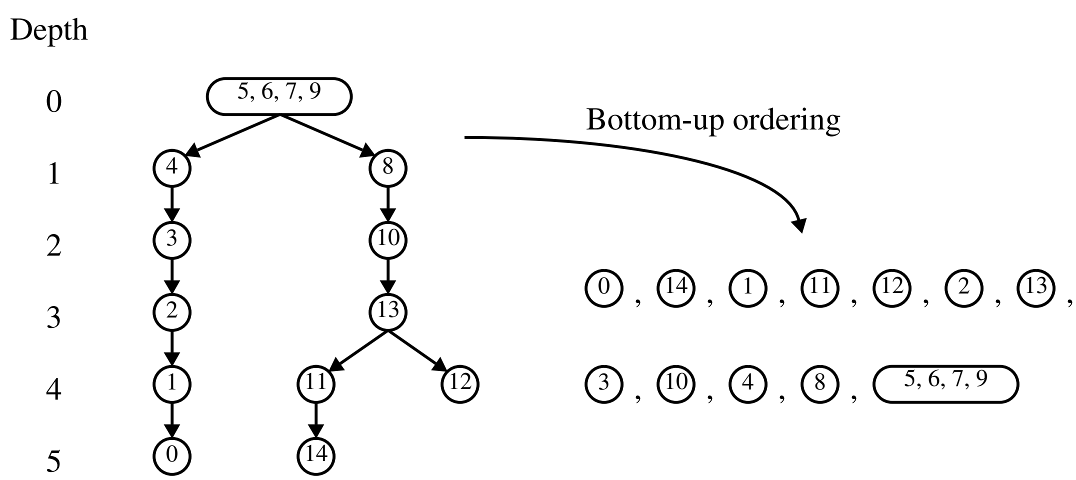

Figure 5.4 Bottom-up supernode ordering creating by sorting according to depth in the elimination tree.

Figure 5.5 shows the algorithm used to analyze the flow of data through the elimination tree using the bottom-up ordering of supernodes. Line 1 of this algorithm initializes all supernodes as neutral (i.e., neither direct nor iterative). Line 2 then loops over the supernodes using the bottom-up traversal of the elimination tree. Within the loop, only neutral supernodes (line 3) are evaluated by the direct/iterative partitioning criteria (lines 4 and 5). For example, one possible criteria computes the memory cost of directly eliminating a supernode. If this cost is too great, the supernode will be marked as iterative. In line 6, if a supernode is marked as iterative, all of its ancestors in the elimination tree will also be marked as iterative by line 7. The result of this procedure is to divide the elimination tree into a lower region to be directly eliminated and an upper region to be solved using an iterative method (as in Figure 5.6).

```
1.  mark all supernodes neutral
2.  foreach(supernode in the bottom-up traversal of the elimination tree)
3.      if(supernode is neutral) then
4.          evaluate direct/iterative partitioning criteria
5.          mark supernode according to results of direct/iterative test
6.          if(supernode is iterative) then
7.              mark all ancestors in the elimination tree as iterative
8.          endif
9.      endif
10. endforeach
```

Figure 5.5 Algorithm for determining direct and iterative supernodes.

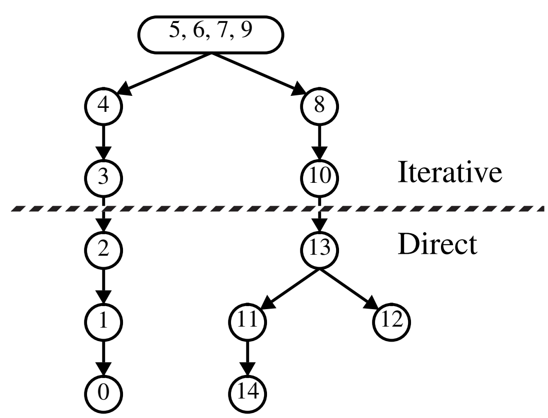

Figure 5.6 Elimination tree partitioned for direct/iterative solution.

Traditionally, symbolic factorization algorithms rely on the underlying assumption that all of the equations are directly eliminated. For the adaptive direct/iterative solver, new techniques which account for the partial elimination must be employed in order to determine the matrix structure. In this case, a snapshot of the matrix structure (i.e., adjacency graph connectivity) after the partial direct elimination is needed in order to profile the structure of the reduced system. The MMD algorithm can be used to create this snapshot by allowing it to symbolically eliminate only the direct supernodes. The resulting adjacency graph represents the structure of the reduced system. Figure 5.7 shows the modified MMD algorithm used to create a snapshot of the connectivity of the reduced system. Although it varies little from the original algorithm presented in Section 4.2.1, three steps have been added. First, line 4 marks the iterative supernodes as ineligible for elimination, thereby preserving them in the adjacency structure corresponding to the reduced linear system. After all of the direct supernodes have been removed and added to the elimination tree, line 14 grafts the iterative supernodes to the top of the elimination tree in order to preserve a consistent representation for the multi-frontal method. Since the iterative method creates no fill-in, a natural ordering of the supernodes is used when they are added to the elimination tree. Finally, line 15 translates the adjacency graph connectivity into an appropriate sparse matrix representation which is used in the remainder of the symbolic computation.

```
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 iterative 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 iterative supernodes to the top of the e-tree
15. preserve connectivity of reduced system for further symbolic computation
```

Figure 5.7 Partial reordering and elimination tree creation for direct/iterative solver.

Figure 5.8 shows the result of applying this modified MMD algorithm to the example grid introduced in Section 4.2.1 using the direct/iterative partitioning shown in Figure 5.6. Extra adjacencies (i.e., fill-in) have been added by the elimination of the direct equations. This matrix structure will be used to determine the storage and indexing for the reduced matrix. The composite elimination tree created by the algorithm is shown in Figure 5.9. The iterative supernodes have been added at the top of the elimination tree using their natural ordering. The top of the tree corresponding to the iterative supernodes exhibits a linear structure. Since no elimination will be performed on these supernodes, any consistent structure will suffice. The composite elimination tree provides a convenient and consistent representation and minimizes the necessary changes to the existing direct solver.

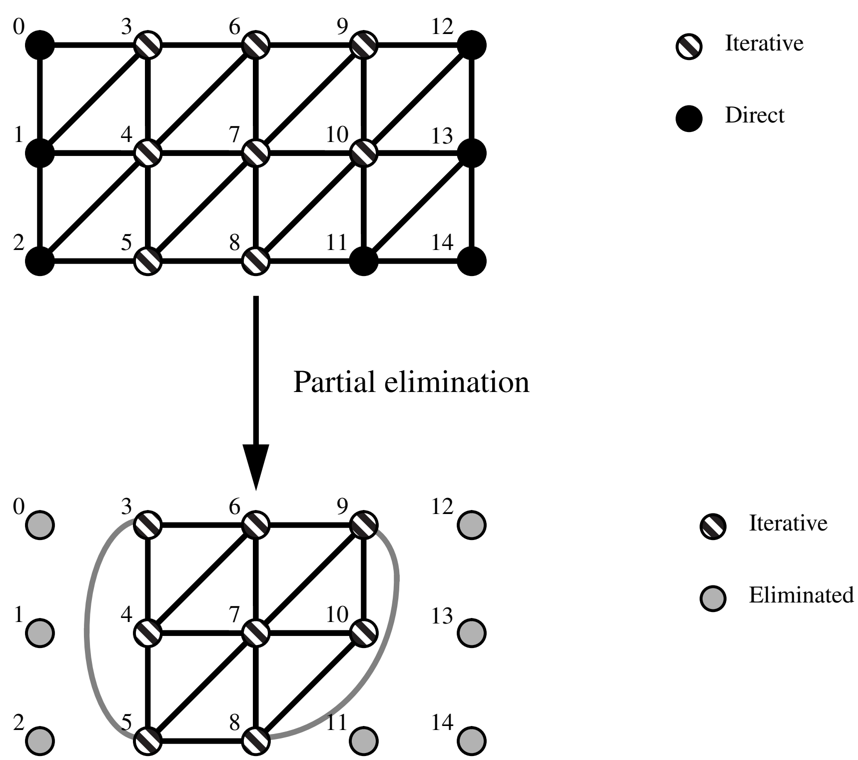

Figure 5.8 Result of performing partial elimination using the MMD algorithm.

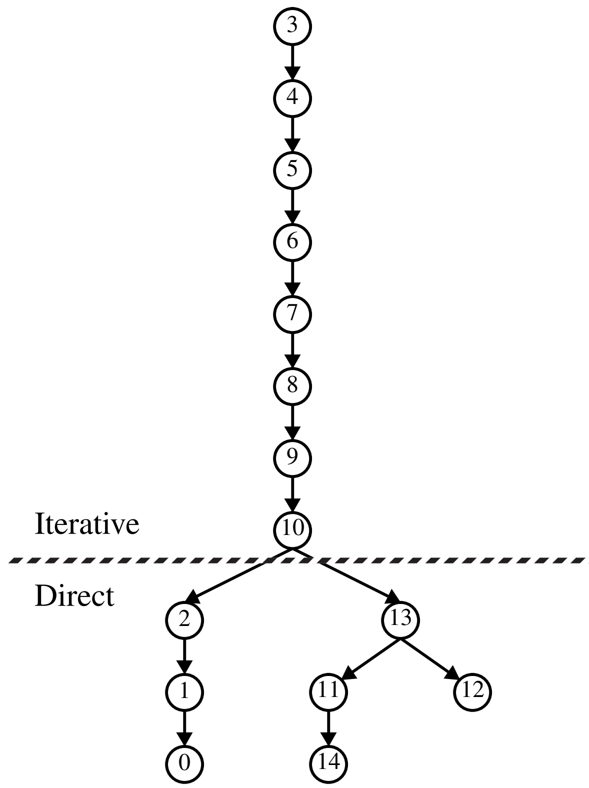

Figure 5.9 Composite elimination tree created for an adaptive direct/iterative solution.

Once the composite elimination tree has been constructed, the traditional functions of the symbolic factorization can be performed. As described in Section 4.2.2, the elimination tree is examined using a depth-first traversal in order to profile the matrix structure. However, the profiling step must be modified to account for the structure of the reduced system created by the partial elimination. Non-zeroes in the matrix structure corresponding to the directly eliminated supernodes can be determined using existing procedures. The adjacency graph for the reduced system is used to create the matrix structure corresponding to the supernodes to be solved iteratively. Subsequently, the necessary indexing and storage arrays can be allocated for both the direct and iterative solvers. During this step, the temporary storage for the direct and iterative solvers can be overlaid to provide significant memory savings.

5.2.2 Partial Factorization

The multi-frontal factorization algorithm must be modified to produce both a partially factored matrix and a reduced system for the iterative solver. The dichotomy of the composite elimination tree makes these changes straightforward. Figure 5.10 shows the modified multi-frontal algorithm. The test in line 2 has been added to insure that each supernode is correctly processed. For supernodes marked for direct elimination, the existing multi-frontal factorization is performed. Upon encountering the first supernode marked for iterative solution, the direct elimination loop is terminated and the outstanding matrix updates are added into the reduced system (line 12).

```
1.  foreach (supernode in the depth-first traversal of the elimination tree)
2.      if(supernode is iterative) then goto line 12
3.      create a temporary dense matrix for the supernode's front
4.      copy the supernode's initial values to the front
5.      foreach(child in the elimination tree)
6.          add fill-in to the front created by this child's elimination
7.      endforeach
8.      eliminate the supernode from the front using direct kernel
9.      copy eliminated equations to factor storage
10.     copy fill-in to temporary storage for use by parent
11. endforeach
12. copy updates to reduced system matrix
```

Figure 5.10 Adaptive multi-frontal elimination algorithm.

5.2.3 Reducing the Residual

Before the iterative solution of the reduced linear system can begin, the corresponding right-hand-side vector must also be reduced to reflect the partial elimination of equations in the original linear system. To reduce the residual, a partial forward elimination is performed using the factored supernodes. The algorithm shown in Figure 5.11 is similar to the traditional column-oriented forward elimination. As in the partial factorization, a test has been added (line 2) to perform the forward elimination only for the supernodes which have been eliminated. Once the iterative equations are encountered, the loop terminates and the reduced residual values are copied into a temporary vector for use by the iterative method (line 8).

```
1. for i = 1 to number_of_equations
2.     if(iterative(i)) then goto line 8
3.     y(i) = b(i)
4.     for j = i+1 to number_of_equations
5.         b(j) = b(j) - y(i) * l(i,j)
6.     endfor
7. endfor
8. copy results to the iterative reduced residual
```

Figure 5.11 Partial forward elimination to reduce the residual.

5.2.4 The Iterative Step

After the creation of the reduced liner system and the reduced right-hand-side vector, an iterative linear method is employed to solve this matrix problem. None of the algorithmic steps before this stage make any assumptions concerning the iterative linear equation solver to be employed. Consequently, any iterative method appropriate to the problem domain may be used. Of course, a matrix-vector multiplication routine as well as preconditioning routines based on the multi-frontal solver's sparse matrix representation will be required by the iterative method.

The iterative solver implemented in PISCES-MP employs the GMRES algorithm first proposed by Saad and Shultz [47]. A clear presentation of the GMRES algorithm can be found in Shakib's thesis [48]. Additionally, a diagonal preconditioner was implemented in the PISCES-MP solver to scale the reduced system.

5.2.5 Recovering the Direct Solution

The final step required to obtain a complete solution is the recovery of the direct solution through the partial backward elimination shown in Figure 5.12. Unlike the bottom-up computations of the partial factorization and the partial forward elimination, the flow of data in the partial backward elimination proceeds top-down. To accommodate this top-down nature, the partial backward elimination algorithm first copies the solution computed by the iterative step for the reduced system (i.e., the supernodes at the top of the elimination tree) to the final solution vector (line 3). These values are subsequently used to compute the solution for the direct equations (lines 5-9).

```
1.  for i = number_of_equations downto 1
2.      if(iterative(i)) then
3.          copy iterative solution to x(i)
4.      else
5.          for j = i+1 to number_of_equations
6.              y(i) = y(i) - x(j) * u(i,j)
7.          endfor
8.          x(i) = y(i) / u(i,i)
9.      endif
10. endfor
```

Figure 5.12 Partial backward substitution.

5.2.6 Serial Results

The simulation of a PN diode was used to characterize the behavior of the adaptive direct/iterative linear solver within PISCES-MP. The characteristics of the simulation mesh and the resulting matrix are summarized in Table 5.2. An IBM RS6000/590 workstation was used for this study. The simulation was repeated six times with a varying number of equations solved iteratively. Direct/iterative partitions ranging from fully iterative to fully direct were tested. Both solution time and memory usage were measured in order to illustrate the trade-offs involved with varying the number of equations solved iteratively. All data used may be found in Appendix C.

Table 5.2 PN diode simulation and matrix statistics.

Number of grid points	16,384
Number of triangles	31,852
Number of equations	49,152
Non-zeroes in matrix	1,022,944
Millions of operations for direct factorization	2,231

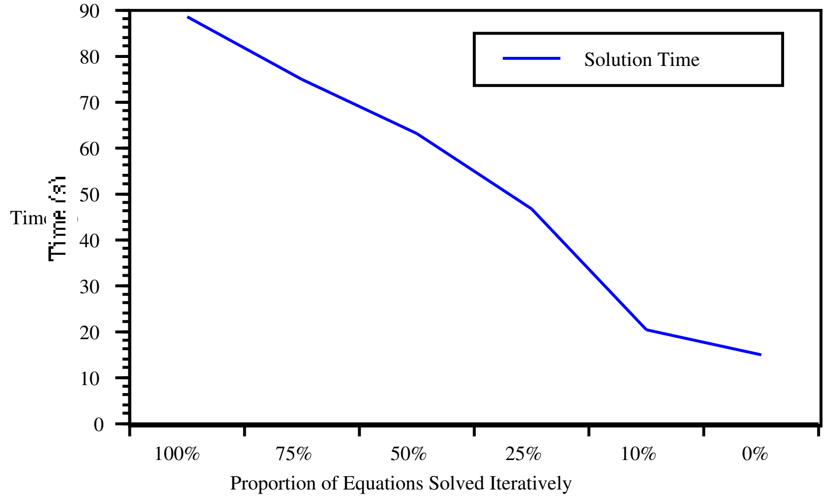

Figure 5.13 Serial solution times for the PN-diode matrix.

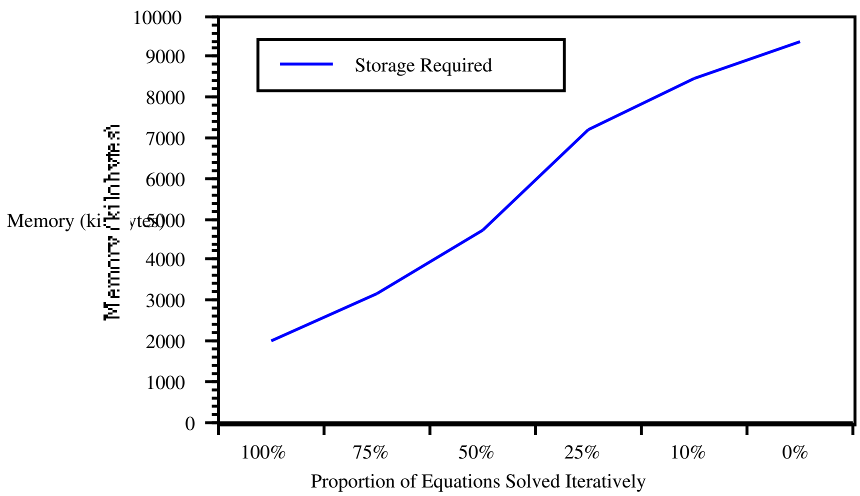

Figure 5.14 Serial memory usages for the PN-diode matrix.

Figure 5.13 shows the linear solution times as the number of equations solved iteratively ranges from 100% to 0% of the total number of equations. The fully iterative solution required the most time while the fully direct solution required the least time. As the number of equations directly eliminated was increased, the solution time also decreased. When 90% of the equations are directly eliminated, the adaptive solution finally becomes competitive with the fully direct solution. This behavior corresponds to the observations of Ferencz [16] that direct solvers provide the fastest solution times for ill-conditioned two-dimensional problems like those produced within PISCES-MP. Figure 5.14 shows the memory usage for each direct/iterative partitioning. As expected, the fully iterative solution requires the least amount of memory, and the memory usage grows as the number of directly eliminated equations increases (and as solution time falls). In effect, by increasing the number of equations directly eliminated, memory usage is being traded for improved solution time. For problems in which memory is the limiting resource, the adaptive approach provides flexibility and allows the solution time to be improved as much as memory resources permit. For example, problems arising from the simulation of fluid-structure interactions (as described in [54]) are often faced with serial memory limitations.

Remark: Contrary to the adaptive solver's behavior in a serial environment, a parallel implementation of this solution method should provide performance superior to the fully direct solver due to the adaptive solver's ability to perform multiple concurrent eliminations coupled with the superior parallel performance of iterative linear equation solvers.

5.3 Parallel Algorithm

The parallel implementation of the adaptive direct/iterative solver is built upon the parallel multi-frontal method presented in Section 4.3. Modifications have been made to the existing algorithms in order to insure the consistency of the reduced system across shared boundaries. In addition, the iterative solver must be parallelized. The following sections describe the modifications necessary to create a parallel adaptive linear solver. Performance results are also presented.

5.3.1 Symbolic Factorization

As in the serial case, the symbolic factorization step must be enhanced in order to account for the partial direct elimination and the structure of the resulting reduced linear system. Since the adaptive method allows an arbitrary balance between the direct and the iterative solvers, processors must communicate in order to insure a consistent solution strategy for the shared equations. First, each processor independently computes a direct/iterative partitioning of its elimination tree including shared supernodes. Then, the processors reach agreement over the shared supernodes through a communication step. If a shared supernode has been marked as iterative by any processor, all other involved processors must also mark that supernode as iterative. Since the iterative solution requires less memory, it is the preferred solution method when processors disagree over the solution strategy for a shared supernode. Once the direct/iterative partitioning has been computed, the composite elimination tree can be constructed using the existing serial algorithms. Since the ordering of the equations to be solved iteratively is arbitrary, the shared equations can be grouped together in the composite elimination tree in order to improve the performance of the parallel iterative solver.

Using the composite elimination tree, the structures of the partially factored matrix and the reduced linear system are computed. As discussed in Section 4.3.2, processors must communicate to insure a consistent representation of each shared supernode. The direct/iterative dichotomy introduces additional complexity into this process. The processors will communicate not only the structure of the dense matrix blocks corresponding to the directly eliminated shared supernodes but also the sparse matrix structure of the shared supernodes residing in the reduced linear system. After the structures of the partially factorized matrix and the reduced linear system have been established, the existing algorithms can be used to profile and allocate the necessary indexing and storage arrays.

5.3.2 Partial Factorization

The parallel partial factorization algorithm (Figure 5.15) was created in the same fashion as the serial partial factorization algorithm (Figure 5.10). The parallel multi-frontal algorithm (Section 4.3.4) was used as the basis for the algorithm. Factorization of the supernodes to be directly eliminated proceeds unchanged from the existing parallel multi-frontal algorithm. Line 2 has been added to terminate the direct elimination upon detection of the iterative supernodes. Once the partial elimination has ended, line 16 adds the updates created by the partial elimination to the reduced linear system in preparation for the iterative solution step.

```
1.  foreach (supernode in the depth-first traversal of the elimination tree)
2.      if(supernode is iterative) then goto line 16
3.      create a temporary dense matrix for the supernode's front
4.      copy the supernode's initial values to the front
5.      foreach(child in the elimination tree)
6.          add fill-in to the front created by this child's elimination
7.      endforeach
8.      if(supernode is local)
9.          eliminate the supernode from the front using serial kernel
10.     else
11.         eliminate the supernode from the front using parallel kernel
12.     endif
13.     copy eliminated equations to factor storage
14.     copy fill-in to temporary storage for use by parent
15. endforeach
16. copy updates to reduced system matrix
```

Figure 5.15 Parallel partial multi-frontal elimination algorithm.

5.3.3 Reducing the Residual

The computation of the reduced residual (Figure 5.16) is also similar to the parallel forward elimination algorithm presented in Section 4.3.5. The directly eliminated supernodes are used to perform the partial forward elimination. As in the partial factorization, a test has been added (line 2) to terminate execution once all of the direct supernodes have been processed. Line 11 then copies the reduced residual into a temporary vector for use in the iterative solution of the reduced linear system.

```
1.  foreach i = 1 to number_of_equations
2.      if(iterative(i)) then goto line 11
3.      y(i) = b(i)
4.      if(shared(i)) then parallelsum(y(i))
5.      if(owner(i) = me) then
6.          foreach j = i+1 to number_of_equations
7.              b(j) = b(j) - y(i) * l(i,j)
8.          endforeach
9.      endif
10. endforeach
11. copy results to the iterative reduced residual
```

Figure 5.16 Partial parallel forward elimination.

5.3.4 The Iterative Step

As in the serial case, the parallel adaptive linear solver makes no assumptions regarding the iterative method to be employed to solve the reduced linear system. Any suitable parallel iterative solver can be used. However, in order to simply the integration of parallel iterative solvers, an efficient vector globalization routine based upon the equation ordering in the composite elimination tree was implemented.

PISCES-MP uses a parallel implementation of the GMRES algorithm introduced in Section 5.2.4. The parallel GMRES solver is described in Aluru's dissertation [1]. As in the serial implementation, a diagonal preconditioner is used to scale the reduced linear system.

5.3.5 Recovering the Direct Solution

The partial backward substitution used by the parallel adaptive linear solver is shown in Figure 5.17. This algorithm is derived from the parallel backward substitution presented in Section 4.3.5. Line 2 has been added to determine if a supernode has been solved in the iterative step. If so, line 3 copies the iterative results into the solution vector. Otherwise, a traditional row-oriented backward substitution is performed using the partially factored matrix (lines 5-13).

```
foreach i = number_of_equations downto 1
    if(iterative(i)) then
        copy iterative solution to x(i)
    else
        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
    endif
endforeach
```

Figure 5.17 Partial parallel backward substitution.

5.3.6 Parallel Results

The PN diode simulation described in Section 5.2.6 was also used to characterize the behavior of the parallel direct/iterative solver. A series of direct/iterative partitions were run on an IBM SP2 with RS6000/590 processors. In addition to simply varying the proportion of equations solved iteratively, the direct elimination of the local equations and the subsequent iterative solution of the parallel boundaries was also studied. All data used may be found in Appendix C.

Figure 5.18 and Figure 5.19 show the solution time and memory usage, respectively, for the adaptive direct/iterative solver using two processors of the SP2. As in the serial case, the fully direct solution provides the shortest solution time. However, solving the shared equations iteratively is only marginally slower. The similarities to the serial case are not surprising since the number of shared equations is only a small portion of the total equations, and consequently, the direct solver scales well. The memory usage steadily increases as the number of equations solved iteratively diminishes. Solving only the shared equations iteratively requires as much memory as the fully direct solution (as shown in Figure 5.19) since the reduced iterative system consists of the dense block formed by directly eliminating all of the local equations. As in the serial case, increased memory consumption results in reduced solution time.

Increasing the number of processors from two to four yields similar results. Figure 5.20 and Figure 5.21 show the solution time and memory usage, respectively. In this case, directly eliminating the local equations while iteratively solving both the first-level and second-level shared separators provides the fastest solution time. However, both the fully direct solution and solving only the first-level separator iteratively are competitive with the fastest solution time. As in the two-processor case, the memory consumptions of the fully direct solution and the iterative solution of the first-level shared supernode are identical. However, solving the first-level and second-level shared separators iteratively does provide some reduction in memory usage since the final dense block is not created.

Using eight processors for the simulation begins to show the advantages provided by the adaptive linear solver. The fastest solution time (shown in Figure 5.22) is achieved by directly eliminating the local equations while iteratively solving all three levels of shared separators. This distribution of the direct and iterative equations provides more than a 20% reduction in the solution time as well as roughly a 20% reduction in memory consumption (Figure 5.23).

Figure 5.24 and Figure 5.25 show the solution time and memory usage, respectively, for a sixteen-processor simulation of the PN diode. With sixteen processors, the amount of communication within the fully direct method has increased significantly and reduced its parallel efficiency. Once again, the direct elimination of the local equations followed by an iterative solution of the shared equations produces the fastest solution. With this direct/iterative breakdown, the adaptive solver is nearly 2x faster than the fully direct solver and requires only 60% of the memory used by the direct method.

The thirty-two-processor simulation produces even more dramatic results than the sixteen-processor simulation. Figure 5.26 and Figure 5.27 show the solution time and memory usage, respectively. Once again, the direct elimination of the local equations followed by an iterative solution of the shared equations produces the fastest solution. A 5x reduction in solution time and more than a 3x reduction in memory usage over the fully direct solution are achieved.

Finally, Figure 5.28 contrasts the direct solution times with the lowest achieved solution times of the adaptive solver. Clearly, as the number of processors increases, the adaptive solver performs far better than the direct method. In addition, the adaptive solver's memory demands are less than the memory demands of the fully direct solver.

At first, the parallel results appear surprising based upon the serial behavior of the adaptive solver. Contrary to expectations, the increasing numbers of equations solved iteratively as the number of processors increases produces solution times faster than the fully direct approach. However, in the parallel environment, two factors which modify the adaptive solver's behavior have emerged. First, the lower communication demands of the parallel iterative solver lead to superior overall scalability. Secondly, the parallel partitioning distributes the direct elimination more uniformly throughout the simulation grid than the global MMD ordering used by the serial solver. Consequently, the preconditioning effect of the direct elimination is more widespread and more uniform than in the serial case. As more parallel partitions are added, the granularity of the "direct preconditioner" is further refined. This behavior suggests one approach for improving the serial solver: mimicking the equation ordering produced by the parallel partitioning. In some respects, the parallel adaptive solver exhibits similarities to multi-grid solution methods [51].

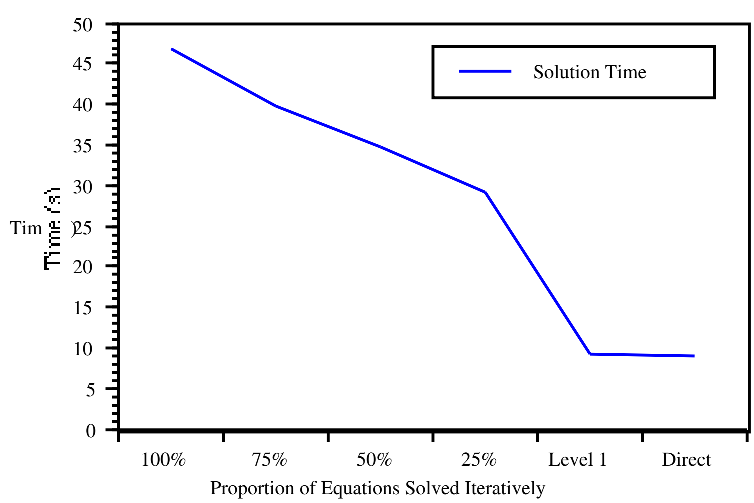

Figure 5.18 Two-processor solution times for the PN-diode matrix.

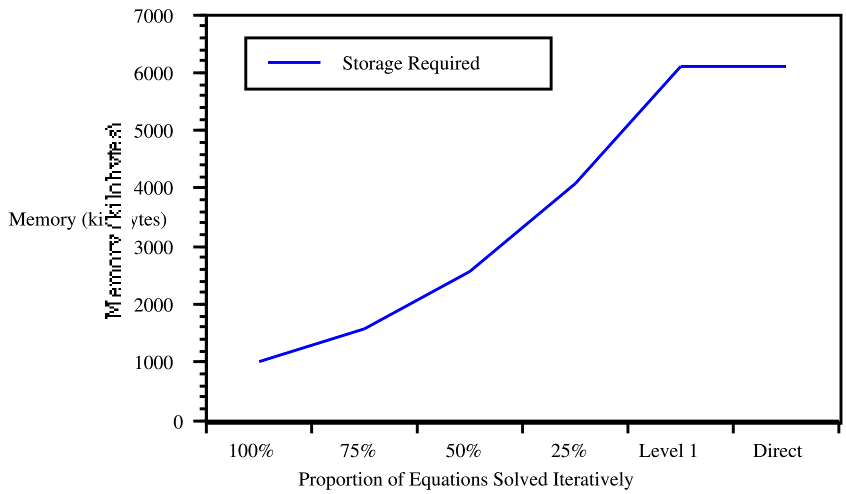

Figure 5.19 Two-processor memory usages for the PN-diode matrix.

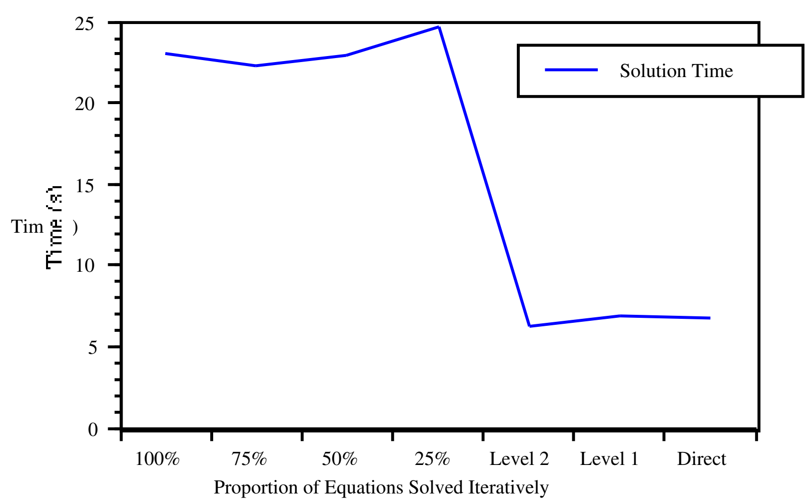

Figure 5.20 Four-processor solution times for the PN-diode matrix.

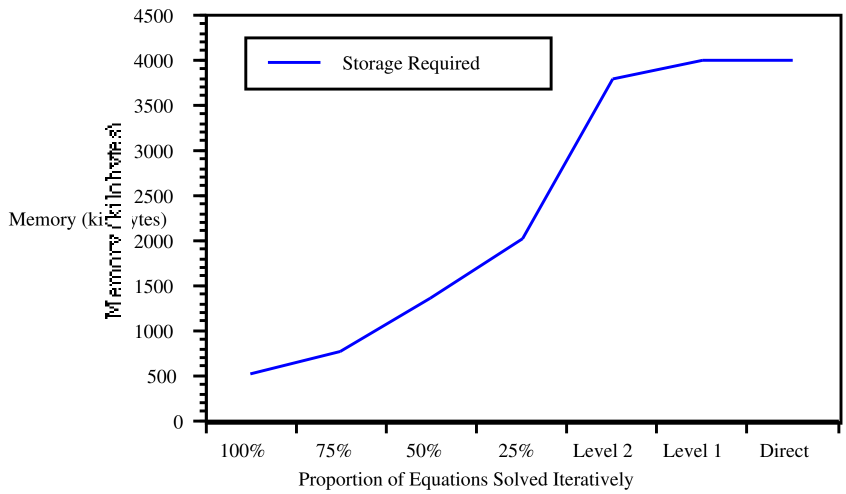

Figure 5.21 Four-processor memory usages for the PN-diode matrix.

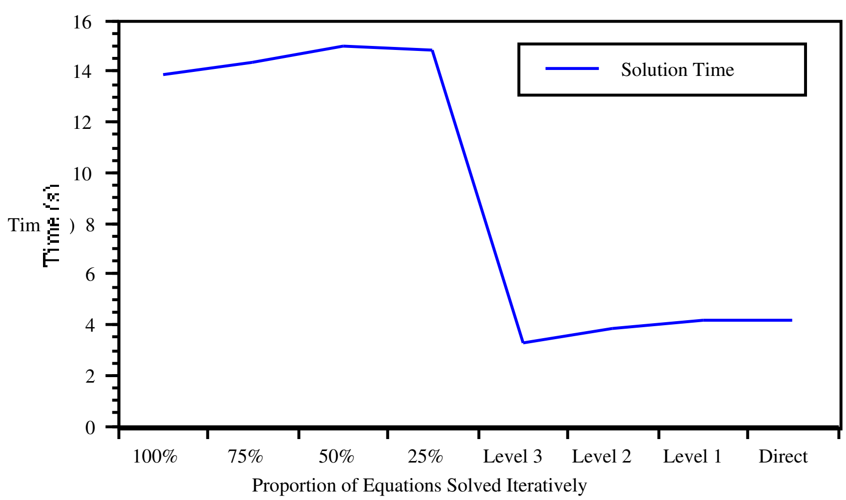

Figure 5.22 Eight-processor solution times for the PN-diode matrix.

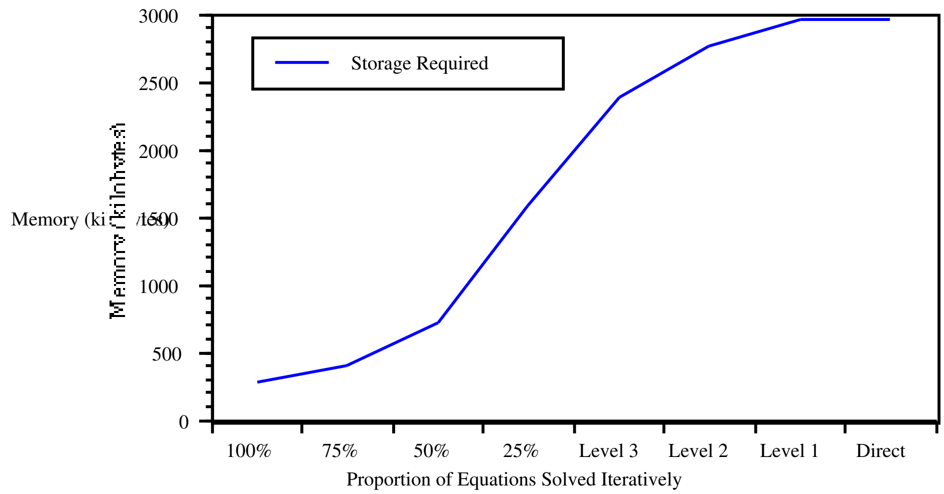

Figure 5.23 Eight-processor memory usages for the PN-diode matrix.

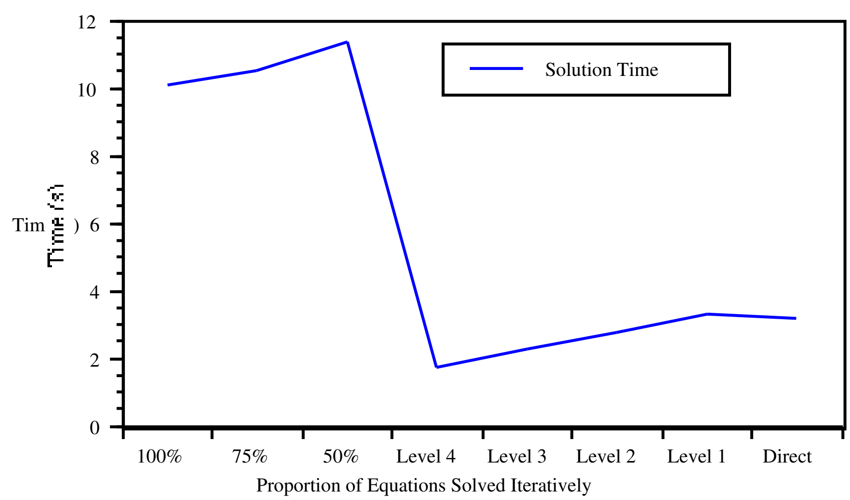

Figure 5.24 Sixteen-processor solution times for the PN-diode matrix.

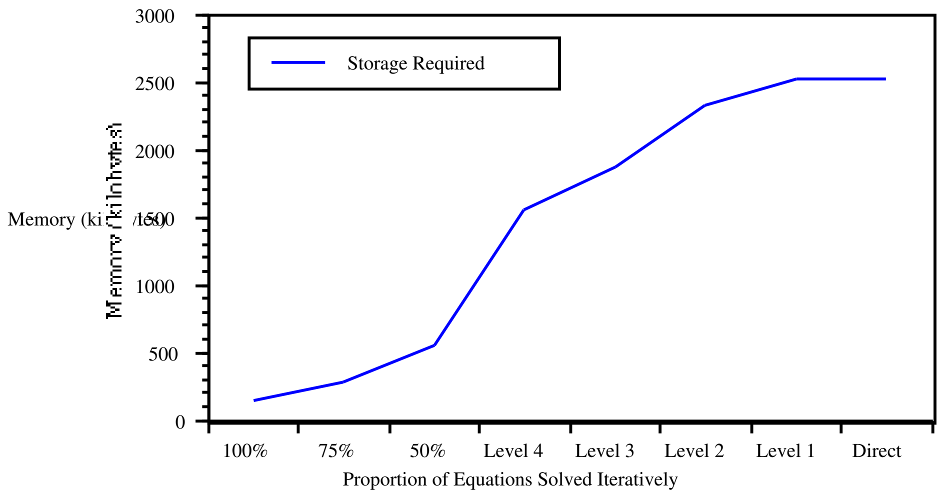

Figure 5.25 Sixteen-processor memory usages for the PN-diode matrix.

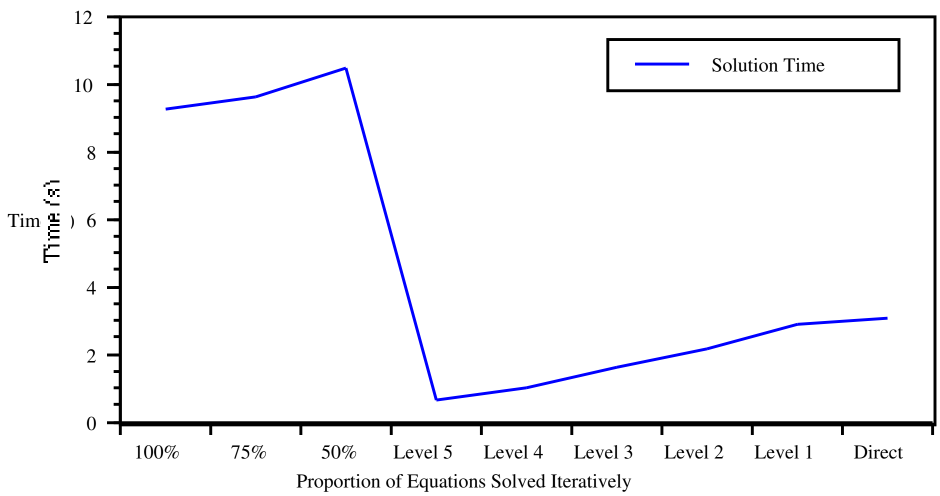

Figure 5.26 Thirty-two-processor solution times for the PN-diode matrix.

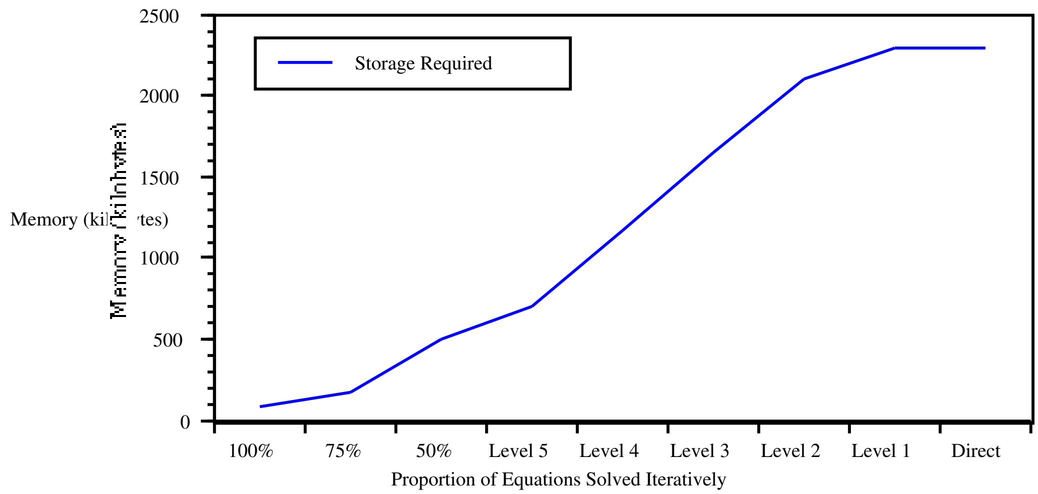

Figure 5.27 Thirty-two-processor memory usages for the PN-diode matrix.

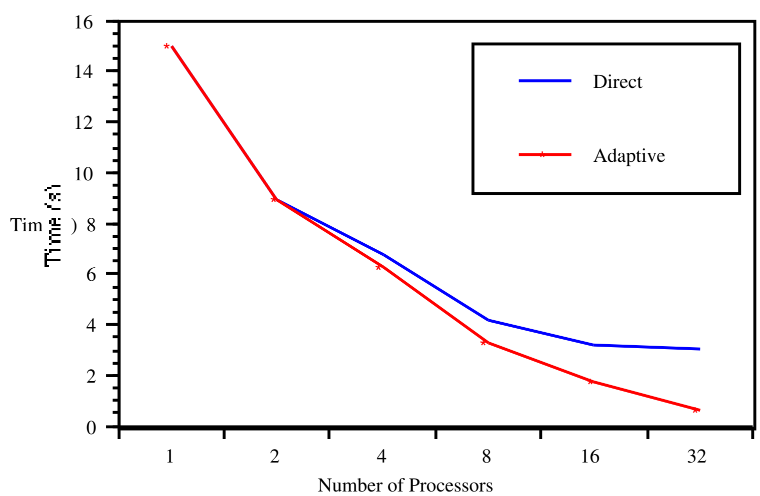

Figure 5.28 Direct vs. adaptive execution times for the PN-diode matrix.

5.4 Summary

An adaptive direct/iterative solution method which is based upon a multi-frontal direct solver and provides greater flexibility than traditional linear solution methods has been presented in this chapter. An augmented symbolic factorization step provides the ability to adaptively choose the breakdown between equations to be directly eliminated and equations to be solved iteratively. Provided with an appropriate representation of the structures of the partially factored matrix and the reduced linear system, the subsequent modifications to the numeric steps are straightforward. Moreover, any appropriate iterative equation solver may be accommodated within this framework, making the adaptive solver suitable to many application areas.

In a serial environment, the adaptive solver provides the ability to efficiently exploit computational resources to solve ill-conditioned problems by balancing the memory demands of direct methods against the convergence properties of iterative methods. In a parallel environment, the adaptive solver can reduce both the memory requirements and the solution times in relation to a fully direct solution. As the number of processors increases, the superior scalability of an iterative method coupled with the more orderly distribution of directly eliminated equations resulting from the parallel partitioning provide excellent improvements over the fully direct solver. Provided with the utility and flexibility of the adaptive direct/iterative method, further study to improve this method is certainly warranted.