Global Preordering for Newton Equations Using Model Hierarchy

Kirk A. Abbott, Benjamin A. Allan, and Arthur W. Westerberg
Dept. of Chemical Engineering and the Engineering Design Research Center, Carnegie Mellon University, Pittsburgh, PA 15213

In solving large simulation problems using Newton's method, a large sparse linear system $Ax = b$ has to be solved. The cost of solving its problems can dominate the overall solution cost of the problem. Current approaches of reducing these costs are reviewed, and then a new algorithm for preordering the sparse matrix A is described that is based on the hierarchical structure provided by an object-oriented description of many recent modeling systems such as ASCEND, gPROMS, DIVA, and Omola. Particularly, rapid preorderings are obtained to support interactive manipulation of models and efficient solutions in automatic process synthesis algorithms, two applications where the preordering cost will be spread over only a few factorizations. With a factorization routine that permits a-priori reorderings (LU1SOL), this algorithm produces order of magnitude reductions in analysis and factoring times as well as in fill and operation count over our previous experience. The time to factor the 50,000 Newton equations for a highly recycled ethylene plant model is of the order of a few seconds on a conventional workstation. Abstracting and applying the fundamental concepts of this algorithm made it possible to improve the performance of the ma28 code significantly. This approach makes solution speeds competitive with and generally more consistent than codes considered the state-of-the-art (ma48 and umfpack 1.0).

Introduction

About three years ago, we started an investigation into how we might improve our equation-based modeling environment ASCEND (Piela et al., 1991) so it could solve much larger problems. The version of ASCEND at that time was capable of solving models up to about 10,000 equations. A 150-megabyte workstation we used ran out of space for larger problems; it also took a few tens of minutes to solve such a problem. Our goal was an order of magnitude increase in problem size while solving on the same workstation as we then used.

We first wished to reduce the solving times, the primary subject of this article. We solve the simultaneous nonlinear equations generated by the model iteratively using a variant of Newton's method. This method has, as an inner loop, the time-consuming repeated solution of a set of linear equations. The coefficient of these linear Newton equations is the Jacobian matrix for the nonlinear equations evaluated at the current guess for the solution. Solving involves factorizing this Jacobian matrix into its LU factors.

Our experience from the early 1980s for Jacobian matrices derived from chemical flowsheet models suggested that factorizing times for them grew quadratically with the number of equations, a growth one would like to avoid. Thus our strategy to get reduced solve times might revolve around keeping the problems we have to factorize small. However, many flowsheets give rise to very large sets of equations that must be solved simultaneously. Our only hope would be to decompose these large simultaneous problems.

To assess the potential for improving factorization times, we ran the following computational experiment. We set up a model in ASCEND involving four distillation columns. It gave rise to about 6,000 equations. The Jacobian matrix had approximately 36,000 nonzeros in it, that is, an average of about 6 nonzeroes per equation. In the first variant of the model, we sequenced the columns by having a product from the first column serve as the feed to the second, a product from the

Correspondence concerning this article should be addressed to B. A. Allan, Dept. of Chemical Engineering, Carnegie Mellon Univ., Pittsburgh, PA 15213, second as the feed to the third, and finally a product from the third as a feed to the fourth. The resulting nonlinear equations had a block lower triangular (BLT) permutation primarily of four large blocks, one for each column. Our solver found and solved these blocks sequentially. However, we turned off the BLT permutation (a flag setting accessible through the interface of ASCEND) and forced the system to treat all the equations simultaneously. The LU factors had about 50,000 nonzeros in them, 14,000 more than in the original Jacobian corresponding to a growth factor of just under 1.4. This is a very modest amount of *fill.*

We then altered the model so a product from the fourth column recycled back to the first. The recycle added only a few nonzeros to the 36,000 already in the Jacobian. These nonzeros changed the behavior when factorizing as the equations no longer had the potential to be permuted to block lower triangular form. One has to solve virtually all 6,000 equations simultaneously. When LU factoring, the number of nonzeros grew to about 360,000, a growth factor of about 10. Our experience suggests such a factor is not uncommon. We wondered how so few nonzeros added to the Jacobian caused such a dramatic change in the growth of nonzeros. This growth has at least three unwanted side effects: much longer factorizing times, a requirement for much more computer memory to factor, and more chance for round-off errors to become a problem. We wondered if we could automatically find and treat as special those few nonzeros that caused the problem and regain the much reduced growth factors for fill of the first version of the model.

This experiment suggested we should look for algorithms that try to find elements to remove so a matrix will partition into smaller problems. The motivation is simple. Suppose we have a very large problem requiring the simultaneous solution of n equations. If we can successfully decompose these equations into k equal size parts and construct the solution from the parts at virtually no cost, the following model shows that times will decrease by a factor of k if problem times grow quadratically with problem size

$$\frac{\text{solve time with decomposition}}{\text{solve time without decomposition}} = \frac{k\left(\frac{n}{k}\right)^2}{n^2} = \frac{1}{k} \qquad (1)$$

Unfortunately, "clustering" algorithms that operate on sparse matrices to find where to decompose them are NP complete and have the reputation of being expensive to execute. For example, Papalombros (1996) estimated it would take about two minutes to analyze 1,000 equations for a modeling system he is developing at the University of Michigan. If we were fortunate and times scaled linearly with problem size, it would take 100 times that for 100,000 equations.

Factorizing is the carrying out of the sequence of pivoting operations of Gaussian elimination. One would like to choose the pivot sequence to minimize fill, tempering one's decisions so the pivots chosen are not too small to use from a numerical point of view. However, the search for the pivot sequence that overall produces the least fill can often take much more time than factorizing with the pivot sequence that causes the most fill. Thus, sparse matrix packages usually approach finding the sequence by using local decision-making. Typically, they use a greedy approach where, at each pivot step, they pick from among the numerically large enough pivots the one that creates the least fill as the next pivot step is executed. These packages make a sequence of best "local" decisions. Unfortunately, such a pivot may be among the worst globally, leading to explosively large fill later. Some packages restrict the search in this local decision process to an arbitrarily short distance within the sparsity pattern, a strategy which can indirectly take advantage of the flat flowsheet structure of units and streams.

Since clustering algorithms that look at the sparsity pattern for a Jacobian matrix are expensive (Sangiovanni-Vincentelli and Chen, 1977), Karypis and Kumar, 1995a,b,c), we looked for another way to find the clustering. We proposed that the model itself, as expressed within a hierarchical modeling system, offers us information about how we ought to cluster equations so there are few variables connecting them. In this article we show that we can use this information in a very simple and fast manner to discover ways to reorder the equations such that when we carry out the factorization step, we get significant reductions in the resulting fill. The approach is a global one rather than a local one.

We can use the insights gained from this work to explain the solving time experiences seen lately in industry where they are optimizing large detailed process models involving a few hundred-thousand equations to improve the control of processes: they are presenting their models to the sparse matrix package in a very special way, and they are restricting how the sparse matrix packages can select pivots. In the final summary and discussion section, we shall use the insights gained here to suggest when and how they might improve even these times by requiring slightly more information from the modeling packages.

The literature on solving large sparse sets of linear equations which influences this work is reviewed. Existing sparsity preserving reorderings, including those that reorder based on tearing, to motivate our approach is discussed in detail. Our new preordering algorithm, TEAR-DROP, is discussed which we motivate with a simple flowsheet and then illustrate with two examples. The results are presented and discussed of numerical tests we carried out on matrices with order up to 80,000 and over 390,000 nonzeros. Algorithms that we propose can make many existing factorization algorithms much faster to be competitive with the best algorithms available. Detailed data are included on the reordering and factorizing performance of the alternatives tested, followed by a summary discussion.

Review for Solving Large Sparse Sets of Linear Equations

We review some of the important aspects of solving large sparse linear systems of equations of the form $Ax = b$, in particular sparse matrix analysis. We examine the decomposition schemes often used to tackle these systems and attempt to explain the reasons for their inconsistent efficiency. The solution process can be considered to have analysis, factoring, and solution phases (Duff et al., 1989). For our discussion, the matrix A is a sparse real $n \times n$ matrix and does not possess desirable properties such as structural and numerical symmetry, positive definiteness, or diagonal dominance. A has τ nonzero entries.

Matrix decomposition techniques have been proposed in the literature to attempt to break the $O(n\tau)$ cost associated with sparse matrix analysis and factorization. The results have been mixed, with most proponents of decomposition resorting to the apology: "Decomposition techniques, though often times slower than dealing with the problem as a whole, are necessary when the problem size becomes large enough." In addition, some of the decomposition schemes are not necessarily numerically stable (see Edie and Westerberg, 1971; Duff et al.,1989).

Very fast algorithms exist for obtaining a block lower triangular form, provided that a maximum transversal exists (Sargent and Westerberg, 1964; Tarjan, 1972). The complexity has been shown to be $O(\tau)$. One uses sparsity preserving reorderings (SPR) either for minimizing fill-in or to confine fill-in when factoring a sparse matrix. We apply the SPR algorithms to the matrix either *a priori* or dynamically during factorization. The Markowitz criterion is a dynamic reordering scheme. Some of the *a priori* techniques that have been developed are found in Hellerman and Rarick (1972), George (1973), George and Liu (1980), Wood (1982), Stadtherr and Wood (1984a), Coon and Stadtherr (1995). We shall not attempt a detailed discussion of SPR algorithms here. These references as well as Chapter 7 of Duff et al. (1989) should be consulted. It suffices to mention that finding an optimal sparse matrix reordering falls into the class of NP-complete problems, thus the reordering algorithms are heuristic.

The complexity of reordering varies with the different algorithms but has been found to be typically $O(n\tau)$. For large matrices, these reorderings become very expensive, while often failing to yield desirable results. If τ is proportional to n, as it often is for process simulation problems, then the reordering algorithms are of $O(n^2)$. Most SPR algorithms have some *greedy* element as a global optimization is prohibitively expensive. Greedy heuristics along with failure of tie breaking rules very quickly lead to bad decisions which propagate and lead to poor reorderings. Empirically, we have observed that an SPR such as SPK1 (Stadtherr and Wood, 1984a) will start to perform poorly on matrices with as few as 900 to 1,200 rows.

Stadtherr and Wood (1984a) give detailed comparisons of several reordering techniques applied to asymmetric matrices. They report that their BLOKS algorithm (one specialized to chemical engineering stream and unit structures), which restricts reordering to particular regions of the matrix, consistently gave the lowest reordering times. The reordering times for SPK1, one of the better general reordering algorithms, on a few irreducible matrices are shown in Table 1. These times are considered to be prohibitive in an interactive modeling or automatic synthesis context. The Markowitz criteria has also been observed to give very long run times on large matrices (Duff and Reid, 1993).

The factoring phase commonly uses some variant of Gaussian elimination (see Stadtherr and Wood, 1984b; Mallya and Stadtherr, 1995; Carmarda and Stadtherr, 1995; Davis and Duff, 1993, and the references contained therein). Some form of pivoting is used by most of the schemes to maintain numerical stability while factoring. Pivoting is a straightforward exercise in some methods and very difficult in others. If a zero or near zero diagonal element is encountered at any stage of the LU factorization algorithm, it is necessary to exchange that element with an element of larger value. Threshold pivoting is a relaxation of partial pivoting that requires at every step the following relationship be satisfied

$$|a_{kk}| \geq u|a_{kj}|, \quad \forall j, \qquad j > k \tag{2}$$

where u is a suitable value in the range $0 < u \leq 1$. (If $u = 1$ one obtains the partial pivoting criterion.)

The solution of problems by tearing is an approach by which we remove a part of the given problem so that the remaining subproblems can be *analyzed* independently (Hajj, 1980). Most workers who have employed tearing have pushed this concept further; they solve the now independent subproblems (frequently on some sort of multiprocessing computer) and then attempt to combine their solutions with the torn-away part in order to obtain the solution for the overall problem (George, 1974; Hajj, 1980; Vlach, 1985; Moriyama, 1989; Iordache, 1990; Nishigaki et al., 1990, 1991; Zecevic and Siljak, 1994). Tearing was apparently first introduced by Kron in 1951 (Kron, 1963). It is a widely used technique in chemical engineering practice applied to solving flowsheets in a *sequential modular* manner (Barclay and Motard, 1972; Upadhye and Grens, 1975; Westerberg et al., 1979) and it amounts to tearing at the nonlinear level of the problem. The success of the tearing techniques has been mixed. Vlach reports higher fill than without tearing. Nishigaki et al. report *much lower* fill. Almost all workers find that there is an *increase* in factorization time when using single processors, and most resort to the *apology of decomposition* we mentioned earlier. A serious issue can be that of the numeric stability of block factorization schemes as aptly raised by Duff et al. (1989), as well as the applicability of the algorithms to large asymmetric matrices. The matrices tested in the literature cited are of relatively low order and are often structurally and numerically symmetric.

Motivation for a new algorithm

We are motivated to develop a new algorithm based on the following observations:

- SPRs are expensive. For large matrices, the analyze time can be significantly greater than the factorization time because of the $O(n\tau)$ or worse complexity. We wish to obtain much better reordering times, in a more general way than that of the BLOKS algorithm.
- SPR are based on heuristics. In particular most of these algorithms employ row and column counts in their decision-making process. They also depend on tie breaking rules. For large problems, or problems with a lot of structure, the tie breaking rules quickly fail. Bad decisions are made which then are allowed to and usually do propagate throughout the rest of the reordering.

Table 1. Reordering Times

	Order n	Nonzeros	Time (s)*
4 Cols	12,456	44,800	20.89
10 Cols	31,140	115,830	119.98
Isom_30K	19,995	105,102	27.73

*HP 9000/715 workstation.

• Explicit tearing should be avoided to reduce the introduction of artificial singularities during solving. Only when memory available on the processor(s) is inadequate should we resort to hiding information by tearing.

Based on our introductory discussion, we make the following conjecture. *The cost of factoring an irreducible matrix A should be only slightly more expensive than that of factoring a bordered block diagonal matrix A′ nearly identical to A, where A′ is highly reducible.*

TEAR-DROP Reordering Algorithm

Our objective is to solve the linear set of equations, $\boldsymbol{Ax} = \boldsymbol{b}$, where matrix A is an irreducible block extracted from the Jacobian matrix for the nonlinear equations of our model. We present here an algorithm to order a sparse matrix that improves subsequent matrix-based reorderings and factorizations. We call our algorithm the TEAR-DROP algorithm (tearing, decomposition, reordering and partitioning). The input to this algorithm is an unacceptably large irreducible matrix A, one that has more than $n_{\max}$ rows and columns in it, where $n_{\max}$ is the maximum size matrix to which we wish to apply a matrix-based SPR algorithm. A typical maximum size is about 1,000 for our flowsheeting examples.

The TEAR-DROP reordering algorithm discovers how to decompose such a matrix and put it into a recursive bordered block diagonal (RBBD) form, as shown in Figure 1. The algorithm makes use of a model part/whole hierarchy in the form of a directed acyclic graph (DAG) and may be applied to any system of equations grouped hierarchically. We shall show

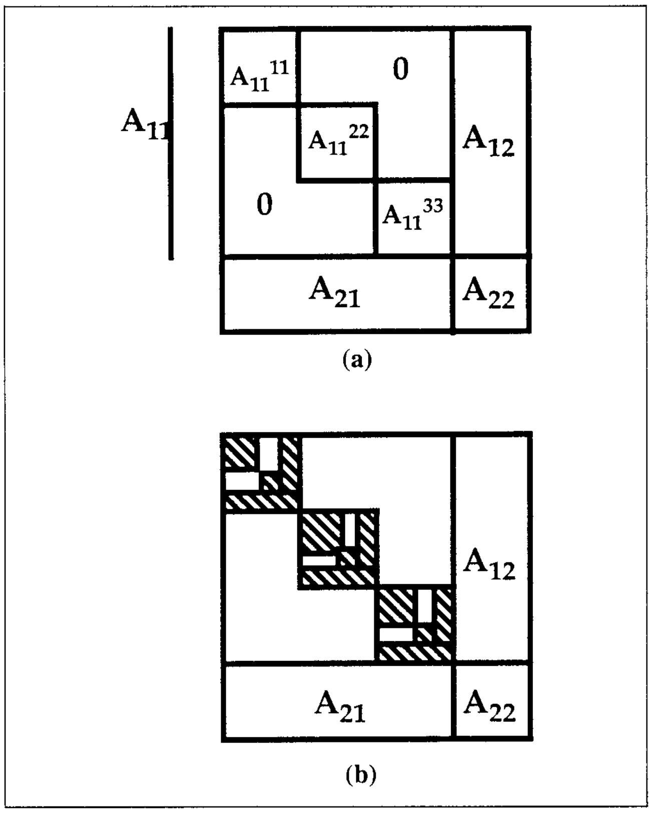


Figure 1. (a) Bordered block diagonal and (b) recursive bordered block diagonal matrices.

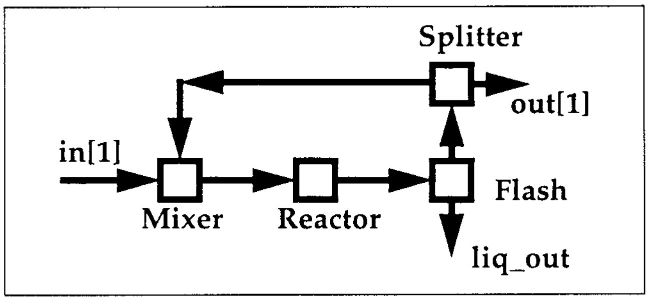


Figure 2. FS, a simple flowsheet.

that the complexity of the ordering algorithm is $O(\tau \cdot \log n)$. When the reordered matrix is submitted to a linear equation solving code to be LU factored, the factoring code used must not apply its own sparsity preserving reordering in a way which will substantially disturb the *a priori* ordering unless very difficult pivots are encountered.

We first motivate our approach with an example. Figure 2 illustrates a simple process flowsheet containing a recycle. Each of these unit models is built of parts, including its input and output streams. For greater generality, we assume the equivalence of an input stream for one unit to an output stream for another. This assumption will lead to a directed acyclic graph (DAG) form of part/whole hierarchy. The streams themselves may also be built of parts, including a part, for example, that includes the equations to compute the physical properties for it.

We may wish to add specification equations which relate variables belonging to more than one unit model. For example, we might add a somewhat idyllic equation in the flowsheet that fixes the flow for the liquid stream leaving the flash unit to equal the flow for a component in the feed to the flowsheet. In a part-whole notation common to many object-oriented languages this equation might be written

$$\text{Flash.liq\_\_out.Ftot} = \text{Mixer.in[1].f[propanol];} \quad (3)$$

Let us suppose that we gather together all the equations for the model and fix an appropriate number of variables so the model has an equal number of equations and variables to be computed. We can then permute these equations into a lower trinagular sequence of irreducible blocks. It is to reorder and solve a large irreducible block in this sequence that we introduce the TEAR-DROP algorithm. The block lower triangular permutation algorithm will find a transversal (output set assignment) for these equations as it orders them. We retain this assignment as we need it in the algorithm.

For greatest generality, we may represent the equations and variables in an irreducible block by using a directed acyclic graph (DAG) that represents the part/whole hierarchy of the model. In simulators that associate each stream's equations with only the unit from which the stream exits as a product, this graph will be a tree. The irreducible block of interest in our model is the one representing the recycle part of the flowsheet. Figure 3 presents the DAG for it. The flowsheet node, which we call "FS," is at the left of the DAG. We shall say that is it the *parent* node for its *children* nodes that corre-

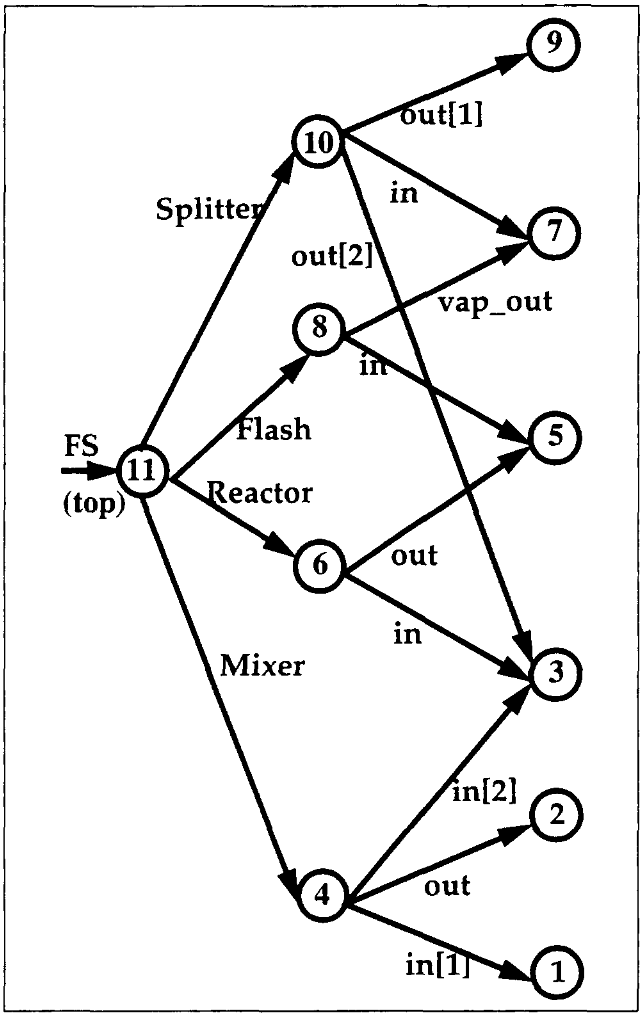


Figure 3. DAG corresponding to the example flowsheet model.

spond to its parts: mixer, reactor, flash, and splitter. Each in turn has children nodes which represent its input and output streams. The structure is a DAG and not a simple tree because nodes have more than one parent when a stream is common between units.

Each node in this DAG introduces new equations into the model. For example, the node for FS introduces (at least) Eq. 3 above to relate the liquid flow out from the flash to the flow of propanol into the mixer. The nodes for the mixer, reactor, flash and splitter models introduce their respective heat and material balances plus any other equations they need. The nodes for the streams introduce the equations needed to compute their physical properties. The names on the links indicate the name by which a child node is known to the linked parent node.

Step 1: preorder the equations

We first traverse this DAG to establish the preliminary order for the equations in the Jacobian matrix A, an irreducible matrix block of a larger problem. We label the nodes in the DAG using a depth first-bottom up traversal algorithm like that which follows.

1. Set k to 1. Unlabel all nodes. Make top node the current node.

2. Descend to a child node with no unlabeled children and make it the current node.

3. Set the label for the current node to k. Enter the equations belonging to the block and coming from this node into the next row positions in the Jacobian matrix. If k is equal to the number of nodes in the DAG, exit.

4. Increment k by one.

5. Retracing the downward path to the current node, ascend to the parent node of the current node. If it has any unlabeled child node, go back to step 2. Else, make this parent node the current node and go back to step 3.

Applying this algorithm to the DAG in Figure 3 can result in the numbering shown (the numbering that results is not unique). Figure 4 shows the preliminary order for the equations in the Jacobian matrix, with the numbered blocks indicating regions of nonzeros corresponding to the nodes in Figure 3. It is on this order for the equations in the Jacobian matrix that we apply our tearing algorithm. Consider the entire matrix to be matrix A. We had no difficulty in making this step of $O(n)$, where there are n rows in A, as it requires us to handle each equation one time as we place it in the Jacobian matrix. This DAG labeling amounts to tearing some edges of the DAG to obtain a tree which then guides the placement of related equations in the matrix such that more closely related questions are placed near each other. The implementation details of this step may vary slightly with a specific modeling system.

Step 2: find the tears

Next divide the matrix A roughly in half horizontally by dividing at a boundary between adjacent equation sets coming from two different nodes. For example, in Figure 4 if all nodes supplied the same number of equations, we might do our splitting at the boundary between the equations supplied by node 5 and those by node 6, as shown. Find the columns with incidence in both halves. Move these columns to the far right of the matrix. Move the rows to which the transversal

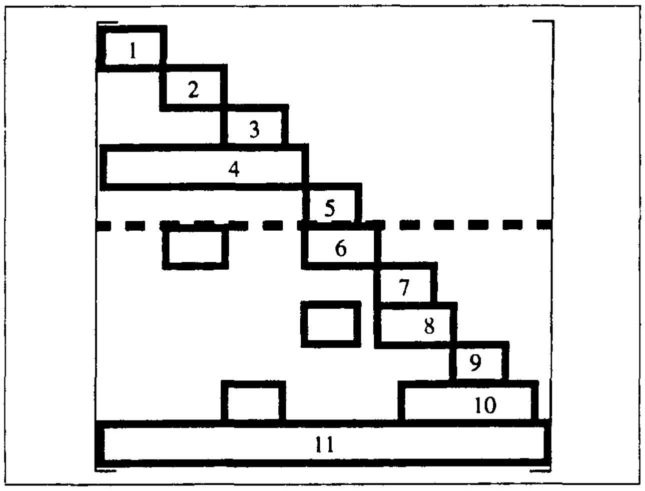


Figure 4. Preliminary equation ordering and first split for the example model FS.

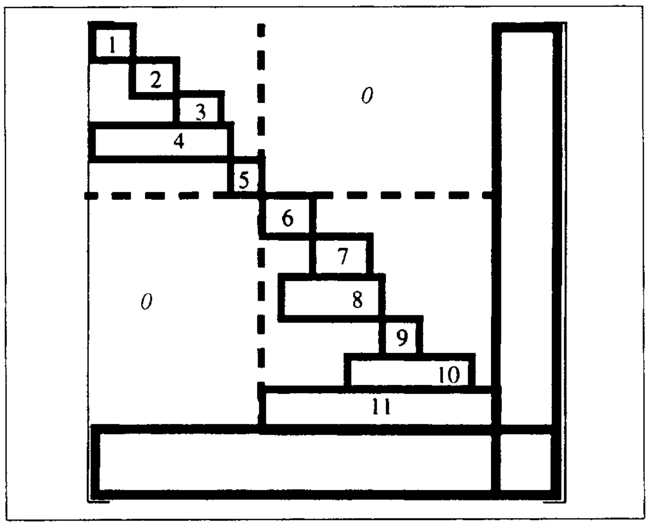


Figure 5. Removing the "tear" columns and rows creates a BBD structure.

algorithm assigned these variables to the bottom of the matrix.

These moves constitute a symmetric permutation of the diagonal, preserving its zero-free character. These columns and rows form the border for the matrix, $A12$, $A21$ and $A22$. The remaining matrix $A11$ is now split into two halves with no variables in common. Placing all the columns for the upper half first and the columns for the lower half second leaves the matrix in bordered block diagonal form (BBD) as shown in Figure 5. Any or all of the blocks 1 through 11 in Figure 4 may be reduced in Figure 5 by the rows and columns moved to the border. This step is $O(\tau)$ as we have to examine each element of the matrix to see in which half it resides.

Step 3: reorder the halves

Ignoring the columns and rows just moved to the border, block lower triangularize the two halves. If any of the partitions is larger than $n_{\max}$, make it matrix A and repeat from step 1. This step is $O(\tau)$ the first time if we use Tarjan's algorithm. The second time, we must apply it to each of the two halves of the matrix $O(\tau/2)$ which is still $O(\tau)$, the third time to four problems of size $n/4$ and so forth. We continue until the diagonal blocks are all less than $n_{\max}$ in size. Each step stays $O(\tau)$. The number of bisections will be k where k is approximately the smallest integer that satisifies

$$n/n_{\max} < 2^k$$

Solving for k gives k to be roughly equal to $\log_2(n) - \log_2(n_{\max})$. This preordering is $O[\tau \log_2(n)]$.

Step 4: apply SPR to each block along the diagonal

All the blocks along the diagonal will now be less than $n_{\max}$ in size (unless one of the nodes by itself in the DAG representation produces more equations than $n_{\max}$). Apply an SPR to each of these matrices if the upcoming factorization reorders only on numeric grounds, or restrict the dynamic reordering during factorization to looking locally at a region small enough to keep it in or very near these diagonal blocks (for example, use NSRCH = 4 with ma28).

Step 5: LU factor the entire matrix

LU factor the *entire* irreducible matrix. If the factoring code cannot find an acceptable pivot from a numerical point of view within the block of the current equation, it should seek a pivot in the closest border. Failing that, it should look at the next closest, and so forth, until an acceptable pivot is found. Any code that looks along a row or column and accepts the first reasonable pivot near the diagonal will carry out this step properly. Looking further along into the borders will add extra fill to the problem; the more to the outside the border is, the more fill one will likely experience. If there is no acceptable pivot, the original problem has numerical problems. The code one uses here should not itself apply sparsity preserving reordering to the overall matrix.

Examples

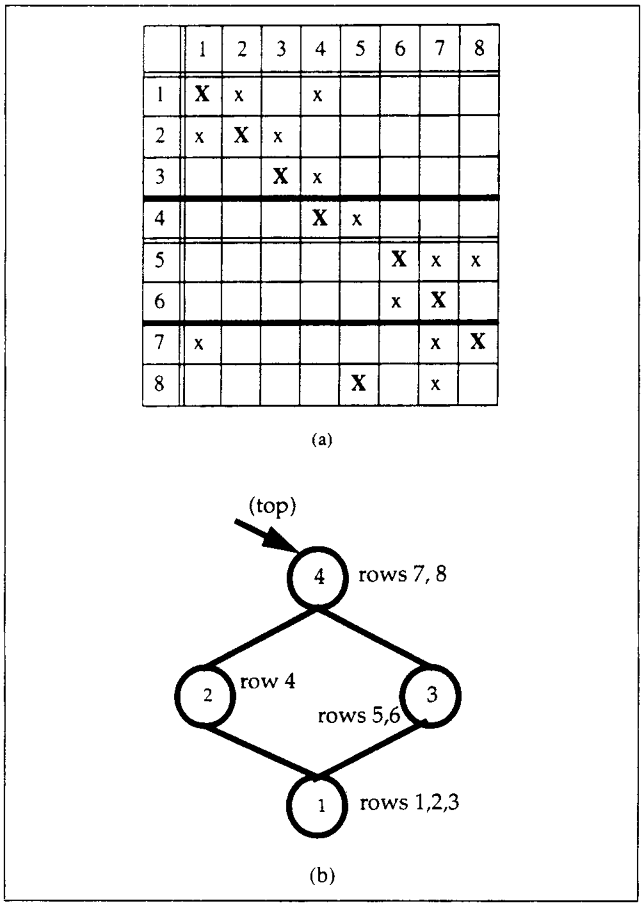
We present two examples here. The first is a very small example involving only eight equations whose purpose is to clarify the algorithm. The second represents a large process model and shows the effect of using the above algorithm to preorder its sparse Jacobian matrix, in contrast to using a sparsity preserving algorithm (SPK1) on the entire matrix.

Eight equation example

We illustrate the ideas with the simple example in Figure 6 involving eight equations in eight unknowns. The large X in each row of Figure 6a represents an output assignment for each equation, found when analyzing the original problem before step 1 of the previous algorithm. These equations form a single partition; that is, they must all be solved together. The DAG in Figure 6b shows how the equations are structured by the model. For example, node 1 at the bottom generates Eqs. 1, 2, and 3. In step 1, we number the nodes in the DAG using the bottom up depth first marking scheme and get the numbering shown inside the nodes. Node 2 (Eq. 4) and node 3 (Eqs. 5 and 6), being on parallel branches, have no variables in common. Variables belong to the lowest node in which they appear in an equation for that node. Thus variables 1 through 4 belong to node 1.

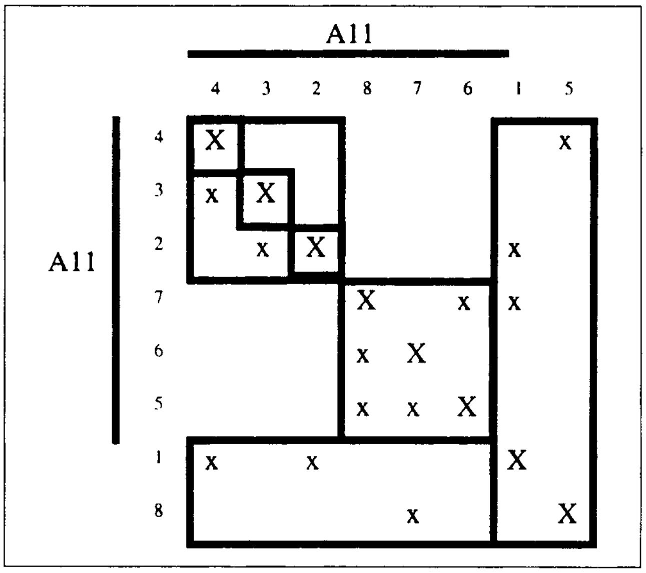
Suppose $n_{\max}$ to be 5 (a small number to allow us to illustrate the algorithm). Then, we must halve the problem as described in Step 2 above. The border we use is between rows 4 and 5 (shown by the double line in Figure 6a), which is the boundary between nodes 2 and 3. Columns 1 and 5 have incidences in both halves. We move these two columns to the far right, and we move the rows to which they are assigned, rows 1 and 8, to the bottom. These form the borders. Rows 2, 3 and 4 form one block; rows 5, 6 and 7 form the other. The first block can be made lower triangular. The second forms a single partition involving three rows and columns. If it is too large, we would recursively split it only for the next step. However, it already has fewer than $n_{\max} = 5$ rows so we stop.

We would next apply a sparsity preserving reordering to only block 2, which is a block with only three rows. This con-



	1	2	3	4	5	6	7	8
1	X	x		x				
2	x	X	x					
3			X	x				
4				X	x			
5						X	x	x
6						x	X	
7	x						x	X
8					X		x	

Figure 6. (a) Incidence matrix and (b) its corresponding tree.



	4	3	2	8	7	6	1	5
4	X							x
3	x	X						
2		x	X				x	
7				X		x	x	
6				x	X			
5				x	x	X		
1	x		x				X	
8					x			X

Figure 7. Partitioned incidence matrix.

trasts with applying it to the entire matrix involving eight rows had we not decomposed the matrix using the TEAR-DROP algorithm. The final result is shown in Figure 7.

4Cols matrix

Figure 8a shows a typical reordering when applying a sparsity preserving reordering to the entire matrix. Figure 8b results from using the RBBD TEAR-DROP algorithm. The matrix in this example is the 4Cols matrix which we shall describe shortly. The reorderings are very different.

Numerical Tests

Test matrices

We tested the algorithm presented in the previous section on a number of matrices, primarily derived from chemical engineering problems. We give the classification and source of the matrices in Table 2. We give the source of the matrix, its order (n), its number of nonzeros (τ), its average nonzeros per row (ρ), the fraction of the rows contained in the largest irreducible block (γ), as well as a brief description for each matrix. The extensive test suite from the Harwell-Boeing collection (Duff et al., 1992) could not be exploited, as part/whole information required to assist the tearing selection is not available. All of the test matrices except rdist3a and 1hr04 were obtained from models developed in the ASCEND system. Information to assist reordering for the rdist3a and 1hr04 matrices was not available.

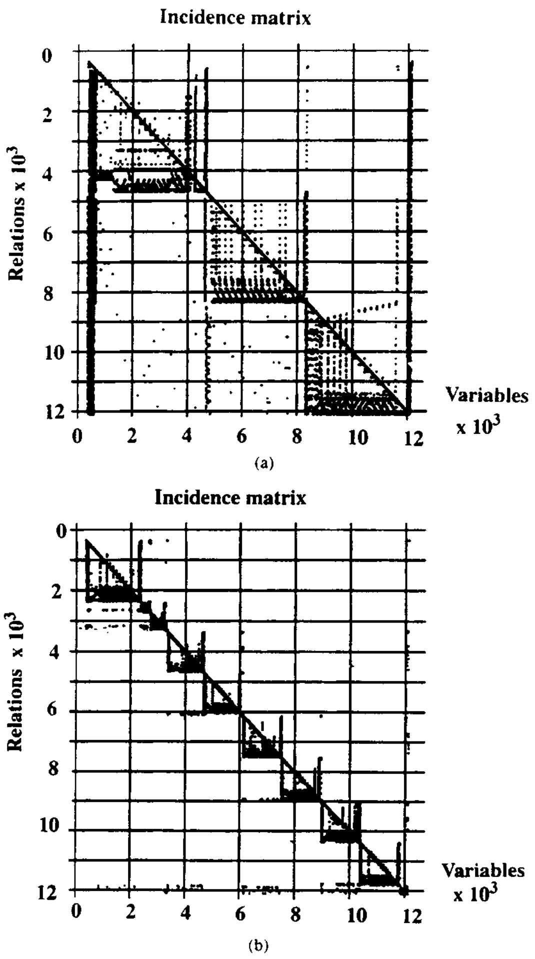


Figure 8. (a) Standard reordering vs. (b) tear-drop reordering.

Table 2. Test Matrices

Case	Identifier	n	τ	ρ	γ	Description
1	Isom__30K	19995	105102	5.256	1.000	Rigorous boundary value formulation of a pentane isomerization reaction with 200 time steps. 4 reacting species in 8 components.
2	4 Cols	12456	44800	3.720	0.945	4 Mass balance distillation columns with 9 components. 30 trays per column, with the last column bottoms recycled to the first column.
3	10 Cols	31140	115830	3.720	0.947	As in 2, but with 10 columns.
4	BigEquil	8986	54389	6.046	0.440	Rigorous distillation column with 30 trays, and 9 components.
5	PPP	14698	64023	4.356	0.558	Rigorous distillation column (propylene splitter) with 164 trays and 3 components.
6	Wood7	6858	33776	4.925	0.512	Complex hydrocarbon flowsheet using rigorous thermodynamics, based on example 7 in (Wood, 1982).
7	Wood8	17509	132081	7.543	0.518	Complex hydrocarbon flowsheet using rigorous thermodynamics and 15 components. Based on example 8 in (Wood, 1982).
8	Ethyl60	59080	294293	4.981	0.343*	Simplified model of ethylene plant.
9	Ethyl80	79554	398163	5.004	0.631	As in 8 but with more stages and different degrees of freedom.
10	rdist3a	2398	61896	25.812	1.000	Reactive distillation column from J. Mallya (1994).
11	1hr04	4101	80755	19.19	0.861	Light hydrocarbon recovery process from J. Mallya (1994)

*Three large blocks, the largest of order 20316.

Codes

In order to test the algorithms proposed, we used five direct (rather than iterative) sparse matrix solvers. These are rankikw, LU1SOL, ma28, ma48 and umfpack1.0. A brief description of the codes tested follows.

rankikw is a multipurpose implementation of the RANKI factorization algorithm and is part of the ASCEND system (Westerberg, 1989). We based it on the work by (Stadtherr and Wood, 1984b). The SPR algorithm is the SPK1 algorithm of Stadtherr and Wood as implemented in the ASCEND system.

Stadtherr and Wood coded *LU1SOL*, a rowwise Gaussian elimination scheme. This code does not save the LU factors but solves immediately for the solution vector x. It was developed in 1984. For the tests, we culled the factorization code from the spar2pas package and fed it with an SPK1 reordered matrix from the rankikw driver.

ma28 (Duff, 1977) is a very popular code from the Harwell Subroutine Library. It uses standard Gaussian elimination with pivoting controlled by the Markowitz criterion. Pivots must satisfy a threshold pivot requirement. The number of rows to be searched in establishing the Markowitz counts can be limited by a parameter NSRCH. This code has many options and only the defaults were used except for the NSRCH parameter. The default value for NSRCH is 32768. The factorization time for this code should be compared with the *total* reorder and factorization for the rankikw and LU1SOL codes because of the Markowitz based pivoting.

ma48 (Duff and Reid, 1993) was designed to replace ma28. It uses columnwise Gaussian elimination with capability to switch to dense code at the later stages of factorization. It is fundamentally different from ma28 in having distinct analyze and factor phases. Like ma28, this code has many options including an NSRCH parameter. The default for NSRCH in ma48 is 3.

umfpack1.0 is the newest of the codes tested and was developed in 1995. It uses an unsymmetric multifrontal algorithm that Davis and Duff (1993) developed and implemented. It uses only dense kernels and uses a dynamic Markowitz search for pivot selection. One can limit the search with an NSRCH parameter as in the case of ma28. The default is 4. It does not use an *a-priori* reordering. The newer umfpack 2.0 was not available when this work was conducted.

The rankikw code is implemented in the C language and can make full use of the dynamic memory allocation features of the language. The other codes are implemented in FORTRAN. ma28, ma48, and LU1SOL had their memory allocated from the C based rankikw driver. We called umfpack1.0 with its own FORTRAN driver. We provided all of the FORTRAN codes with as much memory as possible, a luxury not normally available in interactive simulation environments.

For LU1SOL and rankikw, we report two sets of numbers. The first is the data for these codes using the standard reordering and factorization schemes. The second set are the best times for these codes obtained using the new tearing and reordering (TEAR-DROP) algorithm. We used a value of 1,000 or 2,000 for $n_{\max}$ for the TEAR-DROP algorithm. For ma28 and ma48, Abbott (1996) reports three sets of raw data. These correspond to values for NSRCH of 4, 10 and 1,000, respectively. During preliminary runs with ma28 using the default NSRCH parameter of 32,768, the run times were very long. We decided to investigate the effect of the NSRCH parameter on the ma28 and ma48 codes. However, for the data summaries here, we give only the data for NSRCH = 4. Both codes can be very sensitive to small changes in NSRCH values less than 100, giving inconsistent performance.

The timing results are CPU times in seconds obtained on an HP9000/715 with a maximum available memory of 256 MB. Wherever available, we report the following information

- reordering time (CPU s)
- factoring time (CPU s)
- the number of elements in the factored matrix.

Test results

In comparing the results, the eventual application of the codes is important. In the context of solving an $n \times n$ system of nonlinear equations using Newton's method or a variant, it is well known that approximately seven to ten Newton iterations are normally required for solution when starting from a good initial point. From an excellent initial point (as in slightly perturbed systems, which occurs during numerical integration), one can achieve convergence in as few as two to three Newton iterations. For the former case, the cost of an SPR can be amortized over the seven to ten steps. Due to the significant changes in data over a small number of iterations, it is unlikely that one can use the pivot sequence between factorizations. For this problem class, codes that use *only* dynamic pivoting, such as ma28 and umfpack1.0, are less attractive. Similarly, process synthesis using any form of generation and test scheme requires inexpensive reordering. In the case of sequential numerical integration with many iterations or long running real-time optimization models, the cost of reordering may be negligible compared to the total factorization time. The slowly changing nature of data also suggests that a pivot sequence may safely be reused (with appropriate safeguards), which favors a code that has these capabilities.

Table 3 shows the performance of the LU1SOL code with the standard SPK1 reordering and the TEAR-DROP algorithm. The TEAR-DROP times include all the time taken for tearing and reordering. In all cases the analysis and factor times are faster with the TEAR-DROP algorithm, in some instances by an order of magnitude. The number of fills and the operation counts are also always lower. The SPK1 algorithm is used as the SPR in TEAR-DROP Step 4.

Table 3. Normal Reorder vs. Tear Drop

	SPK1 and LU1SOL			TEAR-DROP and LU1SOL		
	Reorder	Factor	Total	Reorder	Factor	Total
Isom__30K	27.73	16.35	44.08	5.29	0.87	6.16
4Cols	20.89	22.39	43.28	2.53	2.93	5.46
10Cols	119.98	114.78	234.76	9.81	7.27	17.08
BigEquil	1.9	4.7	6.60	0.46	0.49	0.95
PPP	10.27	0.41	10.68	1.9	0.38	2.28
Wood7	1.5	0.24	1.74	0.42	0.18	0.60
Wood8	13.38	26.47	39.85	2.34	4.36	6.70
Ethyl60	50.67	13.22	63.89	8.78	2.31	11.09
Ethyl80	204.94	32.35	237.29	15.49	3.95	19.44

We obtained similar results for the rankikw code (see Table 4). However, the improvement in factorization times was not as significant nor as consistent as for the LU1SOL code. This is largely due to an expensive relabeling operation in the rankikw implementation, which subsequent work in other areas has eliminated.

Table 4 gives the analysis and factor times for the five codes tested. In this table the total analysis and factor times are reported. For the codes which have a distinct analysis phase (rankikw, LU1SOL and ma48), the analysis time is provided in parentheses. We do not intend a direct comparison of the codes, but perhaps the comparison is inevitable. Ignoring the tested implementation of the rankikw code, all codes are within a factor of 4 of the fastest code, and which code is fastest varies with the problem.

One conclusion that may be drawn is that the TEAR-DROP algorithm can make an older code competitive with state-of-the-art codes if it can take advantage of better *a priori* reorderings. Another conclusion is that TEAR-DROP is using explicit information about problem structure while some matrix-based algorithms are either implicitly attempting to infer the structure or explicitly assuming a matrix with a block tridiagonal structure which is common but not universally present in chemical engineering problems.

Hajj (1980) defines tearing as *an approach by which part of the given problem is torn away so that the remaining subprob-*

Table 4. Analyze and Factor Time Summary*

	rankikw and TEAR DROP	LU1SOL and TEAR DROP	ma28 NSRCH = 4	ma48 NSRCH = 4	umfpack1.0
Isom__30K	(5.45) 46.06	(5.45) 6.16	4.56	(1.32) 1.67**	2.64
4Cols	(2.91) 12.48	(2.53) 5.46**	6.47	(6.28) 8.24	5.97
10Cols	(9.81) 61.61	(9.81) 17.08**	18.85†	(22.65) 29.49	18.35
BigEquil	(0.49) 3.25	(0.46) 0.95	0.88	(0.39) 0.54**	1.59
PPP	(1.89) 9.62	(1.90) 2.28	1.42	(0.45) 0.59**	2.08
Wood7	(0.31) 1.41	(0.42) 0.60	0.29**	(0.25) 0.34	1.10
Wood8	(2.34) 15.74	(2.34) 6.70	3.44**	(2.79) 3.62	5.03
Ethyl60	(7.14) 60.51	(8.78) 11.09	5.69	(2.52) 3.22**	7.47
Ethyl80	(15.65) 286.3	(15.49) 19.44	N/A	(4.24) 5.25**	9.87

*The numbers in parentheses are the *a priori* analyze times where applicable.
**Shortest time for the named matrix.
†ma28 with NSRCH = 10, actually had a lower factorization time.

Table 5. Effect of NSRCH on ma48

	1hr04			rdist3a		
NSRCH	Analyze	Factor	Total	Analyze	Factor	total
4	35.38	20.99	56.37	53.07	33.66	86.73
10	33.49	18.49	51.98	49.54	26.63	75.84
40	29.14	15.73	44.87	15.86	38.53	54.39
70	24.43	15.5	39.93	21.68	31.14	52.82
100	15.72	31.48	47.2	14.25	33.77	48.02
1,000	62.27	7.12	69.39	62.24	20.38	82.58

lems can be analyzed independently. By restricting analysis to a few rows via the NSRCH parameter, the Markowitz based codes are practicing *implicit tearing.* A simplistic interpretation of a restricted Markowitz scheme is that the matrix is divided into k independent parts, where $k = n/\text{NSRCH}$. The savings in analysis time is then that given by Eq. 1. The simple heuristic of restricting searches to a few rows or columns has been very successful on the problems tested. However, we have also obtained very inconsistent results, as shown for the 1hr04 and rdist3a matrices (see Table 5).

A surprising result is the much higher fill that is incurred with the *a priori* reordering schemes using the SPK1 reordering algorithm as compared to the dynamic reordering schemes of ma28, ma48, and umfpack shown in Table 6. This same trend is observed when comparing operation counts. This suggests that the SPK1 algorithm may not be as efficient as originally thought, even when restricted to reordering blocks of size $n_{\max}$. A definite extension of this work is re-examination of the above results with different reordering algorithms for the diagonal blocks. This could include the minimum degree algorithm and even a *static* version of the Markowitz criteria; the reordering algorithm would assume that no numerical pivoting would take place and, at any given stage of the reordering, should be made to operate on a region with some maximum size $n_{\max}$, which would need to be determined.

A potential area of research lies in total re-evaluation of the SPR technology. A careful examination of the work by Stadtherr and Wood (1984a) shows that some of the reordering schemes developed were not tested because of their prohibitively long execution times. If, however, we can control the cost of reordering by restricting it to certain regions of a matrix, as is done in this work, then we may examine more sophisticated reordering schemes. The potential exists for research into an entire class of reordering algorithms where each algorithm aggressively seeks to minimize operation counts but which should *never* be invoked on matrices above a threshold size.

Detailed testing of TEAR-DROP algorithm

In the previous section, we examined the performance of a number of different sparse codes that make use of better *a priori* reorderings. In those tests we saw that the TEAR-DROP algorithm used in conjunction with the SPK1 reordering performed well. In this section, we give some more details concerning the performance of the TEAR-DROP algorithm itself.

The scope of these tests is limited. In particular, we used only the SPK1 SPR. One should interpret the kind of results reported here as the basis for determining reasonable values $n_{\max}$ for one solver/SPR combination. One can also use them to evaluate in more detail the reduction in reordering times we presented in the previous section. The data we report in Table 4 are the *total* ordering and SPR execution times. Table 7 presents a breakdown of these times. One can use the ordering times to evaluate the claimed complexity of the TEAR-DROP algorithm.

We used the LU1SOL solver for all the tests wherever we report factorization times. The test matrices we used were those presented in the previous section. For each matrix we tested we provide the following information:

- Ordering time with TEAR-DROP
- Total ordering and SPR time
- Number of diagonal blocks (equal to the number of SPR reorderings done)
- Size of the largest diagonal block
- Number of border "tear" columns found
- Factorization time in seconds on an HP 9000/715.

Summary and Discussion

Inspired by BLOKS and results from tearing experiments, we have developed a matrix preordering algorithm which uses global part/whole information derivable from object-oriented simulators. (As such, this preordering is not specific to process flowsheeting models.) We have seen that this preordering substantially reduces matrix ordering and factoring times on a range of flowsheeting problems. Our approach can handle flowsheets containing units with widely varying numbers of equations per unit. Our approach also addresses concerns expressed by other authors when describing matrix-based algorithms which expect an underlying block tridiagonal structure

Table 6. Nonzeros in Factors

	rankikw and TEAR-DROP	LU1SOL and TEAR-DROP	ma28*	ma48	umfpack1.0
Isom__30K	1.95×10^5	3.08×10^5	1.08×10^5	2.36×10^5	1.23×10^5
4Cols	4.04×10^5	6.00×10^5	9.67×10^4	3.23×10^5	3.68×10^5
10Cols	1.07×10^6	1.43×10^6	2.48×10^5	8.67×10^5	1.05×10^6
BigEquil	1.07×10^5	1.45×10^5	3.30×10^4	7.23×10^4	8.81×10^4
PPP	1.40×10^5	1.24×10^5	4.50×10^4	8.85×10^4	9.10×10^4
Wood7	6.53×10^4	7.82×10^4	6.76×10^4	4.85×10^4	5.78×10^4
Wood8	4.00×10^5	5.04×10^5	9.89×10^4	2.70×10^5	3.42×10^5
Ethyl60	1.06×10^6	9.08×10^5	2.61×10^5	4.35×10^5	4.11×10^5
Ethyl80	1.76×10^6	1.04×10^6	N/A	6.14×10^5	5.32×10^5

*ma28 has the least fill except for matrices Wood7 and Ethyl80.

Table 7. Statistics with RBBD TEAR-DROP

Matrix and Order/No. of Models	$n_{\max}$	Order Time (s)	Total Reordering (s)	No. of Diag. Blocks Reordered	Largest Diag. Block	No. of Border Columns ("tears")	Factor Time (s)
Isom__30K	1,000	4.97	5.22	356	995	2,592	0.85
19995/14201	2,000	4.56	5.33	346	1,995	2,532	0.86
	19,995	0	27.73	1	19,995	0	16.45
4Cols	1,000	1.66	2.49	22	993	1,330	2.85
12456/1482	2,000	0.87	2.86	9	1,997	364	3.08
	11,770	0	20.42	1	11,770	0	22.82
10Cols	1,000	7.48	10.03	53	998	3,081	7.33
31140/3702	2,000	4.26	9.44	26	1,965	1,040	7.62
	29,496	0	118.62	1	29,496	0	115.55
BigEquil	1,000	0.27	0.48	17	855	479	0.52
8986/2545	2,000	0.19	0.65	9	1,788	302	0.93
	3,961	0	1.91	1	3,961	0	4.72
PPP	1,000	1.17	1.90	44	986	323	0.35
14698/5780	2,000	0.64	2.21	16	1,992	111	0.38
	8,201	0	10.27	1	8,201	0	0.41
Wood7	1,000	0.16	0.38	37	891	144	0.19
6858/2284	2,000	0.11	0.54	21	1,677	57	0.19
	3,508	0	1.50	1	3,508	0	0.24
Wood8	1,000	1.49	2.17	19	948	1,017	6.51
17509/4444	2,000	1.09	2.42	16	1,876	744	4.36
	17,509	0	13.51	1	9,087	0	26.47
Ethyl__60	1,000	6.90	8.44	434	995	1,872	2.31
59080/31907	2,000	3.86	7.0	358	1,998	1,272	2.92
	20,316	0	50.67	1	20,316	0	13.22
Ethyl__80	1,000	20.76	22.84	631	997	3,000	3.96
79554/44800	2,000	11.66	15.99	563	1,966	2,164	3.94
	50,172	0	204.94	1	50,172	0	32.34

(Coon and Stadtherr, 1995) by eliminating the need for the block tridiagonal structure.

Many (if not all) of the real-time optimization (RTO) systems in the process industry are equation-based. They use a modular style of architecture in that, for each unit, a system programmer handcrafts a subroutine that can generate the needed partial derivatives and equation residuals for the equations generated by that unit. In addition each unit in these systems generally assumes the responsibility for calling the procedures to evaluate the equations to estimate the physical properties for its output streams. The executive system asks each unit in turn for its equations. It then adds the equations that connect the units and sends all these to a Newton-based solver.

Note that this order of generating the equations is a preordering not unlike that we find for a flowsheet model. If the linear equation solving package for the Newton-based nonlinear solver then restricts pivoting to be local (such as by setting NSRCH to 4 for either ma28 or ma48), the fill will be largely restricted to occur within each of the units first. The connection equations that allow fill to spill over from one unit to another are not used until all the units are pivoted. Our results here suggest these systems will experience reduced fill, as we have. It takes the combination of the ordering used to generate the equations and the limited search when pivoting to get this benefit.

These systems can also directly use the TEAR-DROP algorithm to preorder their equations. They can first construct a DAG (as we did in Figure 3) with the flowsheet as a whole being the top node. Then, they can create a node in the DAG for each unit in the flowsheet and link each as a child node to the top node. Finally, they can create a node for each stream in the flowsheet and attach each as a child to the two nodes that stream connects in the flowsheet. With this DAG, they can readily apply the TEAR-DROP algorithm to establish the order in which to generate the equations for the flowsheet. The order would differ slightly in that streams would be placed closer to the units they connect. The method would yield a type of decomposition similar to what we have obtained using a hierarchical modeling system. The flowsheet is treated as being constructed out of its parts, that is, the units within it. Many embellishments to this module-based algorithm are possible.

However, we suggest that this approach can be improved even more. It does not allow the solver to find substructures in the equations that may be generated by a large unit such as a distillation column. One could modify the column module to view its equations as coming from its parts, which could be column sections and their connecting streams. Each column section could then view its equations as coming from its parts, the trays and their connecting equations. A large unit operation module could represent this additional structural information as an array of integers, one for each matrix row, indicating to which part, subpart, sub-sub part, etc., the row belongs in a fictitious hierarchy of parts. Each unit could pass this information to the solver along with its equations as it generates them. Then, a TEAR-DROP type of algorithm could reorder and decompose the entire flowsheet quickly and from a global perspective, including decomposing the large unit operation modules as needed to gain the full advantage of what we have seen. The units "know" this information; they should supply it to the solver.

Acknowledgments

We wish to thank M. Stadtherr and J. Mallya for the many conversations we had with them and for their generosity in supplying their factoring code and information about it. The member companies of the Computer Aided Process Design Consortium and the National Science Foundation through its grant, No. EEC-8943164, to the Engineering Design Research Center provided the financial support for this project.

Literature Cited

Abbott, K. A., "Very Large Scale Modeling," Dept. of Chemical Engineering, PhD thesis, Carnegie Mellon University (1996).

Barkley, R. W., and R. L. Motard, "Decomposition of Nets," *Chem. Eng. J.*, **3**, 265 (1972).

Barton, P. I., "The Modeling and Simulation of Combined Discrete/Continuous Processes," PhD thesis, Dept. of Chemical Engineering, Imperial College of Science, Technology, and Medicine, London (1992).

Camarda, J. V., and M. A. Stadtherr, "Frontal Solvers for Process Simulation—Local Row Ordering Strategies," AIChE Meeting, Miami (1995).

Coon, A., and M. A. Stadtherr, "Generalized Block-Tridiagonal Matrix Orderings for Parallel Computation in Process Flowsheeting," *Comp. Chem. Eng.*, **19**, 787 (1995).

Davis, T. A., and I. S. Duff, "An Unsymmetric Multifrontal Method for Sparse LU Factorization," Technical Report TR-93-018, Computer and Information Sciences Dept., University of Florida, Gainesville, FL (Mar. 1993).

Duff, I. S., A. M. Erisman, and J. K. Reid, *Direct Methods for Sparse Matrices*, Oxford University Press, New York (1989).

Duff, I. S., R. G. Grimes, and J. G. Lewis, "User's Guide for the Harwell-Boeing Sparse Matrix Collection, Release I," Technical report, Rutherford Appleton Laboratory (1992).

Duff, I. S., and J. K. Reid, "MA48, a Fortran Code for Direct Solution of Sparse Unsymmetric Linear Systems of Equations," Technical report, Rutherford Appleton Laboratory, Oxfordshire, U.K. (1993).

Duff, I. S., "MA28—a Set of Fortran Subroutines for Sparse Unsymmetric Linear Equations," Report r8730, AERE, HM-SO, London (1977).

Edie, F. C., and A. W. Westerberg, "A Potpourri of Convergence and Tearing," *Chem. Eng. Comput.*, **1**, 35 (1971).

George, A., "Nested Dissection of a Regular Finite-element Mesh," *SIAM J. Numer. Anal.*, **10**, 345 (1973).

George, A., "On Block Elimination for Sparse Linear Systems," *SIAM J. Numer. Anal.*, **11**, 585 (1974).

George, A., and W. H. Liu, "A Fast Implementation of the Minimum Degree Algorithm Using Quotient Graphs," *ACM Transactions on Mathematical Software*, **6**, 337 (1980).

Hajj, I. N., "Sparsity Considerations in Network Solution by Tearing," *IEEE Trans. on Circuits and Systems*, **27**, 357 (1980).

Hellerman, E., and D. C. Rarick, "The Partitioned Preassigned Pivot Procedure (p4)," D. J. Rose and R. A. Willoughby, eds., in *Sparse Matrices and their Applications*, Vol. 67–76, Plenum Press, New York (1972).

Iordache, M., "On the Analysis of Large-Scale Circuits," *Bull. Scientific Electrical Eng.*, **52**, 71 (1990).

Karypis, G., and V. Kumar, "A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs," Technical Report 95-035, University of Minnesota, Departmentt of Computer Science (1995).

Karypis, G., and V. Kumar, "Metis: Unstructured Graph Partitioning and Sparse Matrix Ordering System, Version 2.0." Technical report, Dept. of Computer Science, University of Minnesota (1995).

Karypis, G., and V. Kumar, "Multilevel *k*-way Partitioning Scheme for Irregular Graphs," Technical Report 95-064, Dept. of Computer Science, University of Minnesota, (1995).

Kron, G., Diakoptics, Macdonald, London (1963).

Mallya, J., Personal Communication (1994).

Mallya, J. U., and M. A. Stadtherr, "A New Multifrontal Solver for Process Simulation on Parallel/Vector Supercomputers," AIChE Meeting, Miami (1995).

Marquardt, W., "An Object-Oriented Representation of Structured Process Models," *Comp. Chem. Eng.*, 165, 5329 (1992).

Moriyama, S., "Large Scale Circuit Simulation," *Trans. IEICE*, E72, 1326 (1989).

Nilsson, B., "Object-Oriented Modeling of Chemical Processes," PhD Thesis, Dept. of Automatic Control, Lund Institute of Technology, Lund, Sweden (1993).

Nishigaki, M., T. Nobuyuki, and H. Asai, "Hierarchical Decomposition for Circuit Simulation by Direct Method," *Trans. of IEICE*, E73, 1948 (1990).

Nishigaki, M. T. Nobuyuki, and H. Asai, "Availability of Hierachical Node Tearing for MOS Circuits," *Trans. IEICE (Japan)*, J74A, 1176 (1991).

Papalombros, P., Personal Communication (1996).

Piela, P. C., T. G. Epperly, K. M. Westerberg, and A. W. Westerberg, "ASCEND: An Object Oriented Computer Environment for Modeling and Analysis. Part 1—The Modeling Language," *Comput. Chem. Eng.*, **15**, 53 (1991).

Sangiovanni-Vincentelli, A., and Chen Li-Kuan, "An Efficient Heuristic Cluster Algorithm for Tearing Large-Scale Networks," *IEEE Trans. Circuits and Systems, 24, 709–717 (1977).*

Sargent, R. W. H., and A. W. Westerberg, "Speed-Up in Chemical Engineering Design," *Trans. Inst. Chem. Engrgs.*, **42**, 190 (1964).

Stadtherr, M. A., and E. S. Wood, "Sparse Matrix Methods for Equation-Based Chemical Process Flowsheeting I, Reordering Phase," *Comp. Chem. Eng.*, **8**, 9 (1984a).

Stadtherr, M. A., and E. S. Wood, "Sparse Matrix Methods for Equation-Based Chemical Process Flowsheeting II, Numerical Phase," *Comp. Chem. Eng.*, **8**, 19 (1984b).

Tarjan, R. E., "Depth-First Search and Linear Graph Algorithms," *SIAM J. Computing*, **1**, 146 (1972).

Upadhye, R. S., and E. A. Grens, "Selection of Decompositions for Process Simulation," *AIChE J.*, **21**, 136 (1975).

Vlach, M., "LU Decomposition and Forward and Backward Substitution of Recursively Bordered Block Diagonal Matrices," *IEEE Proceedings*, 132 Pt.G(1), 24 (1985).

Westerberg, A. W., H. P. Hutchinson, R. L. Motard, and P. Winter, *Process Flowsheeting*, Cambridge University Press, New York (1979).

Westerberg, K. M., "Development of Software for Solving Systems of Linear Equations," Technical Report EDRC 05-35-89, Engineering Design Research Center, Carnegie Mellon University, Pittsburgh (1989).

Wood, E. S., "Two-Pass Strategies of Sparse Matrix Computations in Chemical Process Flowsheeting Problems," PhD thesis, University of Illinois at Urbana-Champaign, IL (1982).

Zecevic, A. I., and D. D. Siljak, "Balanced Decompositions of Sparse Systems for Multilevel Parallel Processing," *IEEE Trans. Circuits and Systems, I: Fundamental Theory and Applications*, **41**, 220 (1994).

Manuscript received Dec. 19, 1996, and revision received July 21, 1997.