Sparse Partial Pivoting in Time Proportional to Arithmetic Operations*

John R. Gilbert†
Timothy Peierls‡

TR 86-783

September 1986

Department of Computer Science
Cornell University
Ithaca, NY 14853

*Research partially supported by the National Science Foundation under grant DCR-85-14404, and by IBM Corporation under contract 573616.
†Research partially supported by the National Science Foundation under grant DCR-84-51385. Some of this work was done while this author was a visiting professor at the University of Iceland.
‡Research partially supported by the National Science Foundation under a graduate fellowship.

Sparse Partial Pivoting in Time Proportional to Arithmetic Operations*

John R. Gilbert†
Timothy Peierls‡

Computer Science Department
Cornell University
Ithaca, New York 14853

September 1986

Abstract. Existing sparse partial pivoting algorithms can spend asymptotically more time manipulating data structures than doing arithmetic, although they are tuned to be efficient on many large problems. We present an algorithm to factor sparse matrices by Gaussian elimination with partial pivoting in time proportional to the number of arithmetic operations.

Implementing this algorithm requires only simple data structures and gives a code that is competitive with, and often faster than, existing sparse codes. The key idea is a new triangular solver that uses depth-first search and topological ordering to take advantage of sparsity in the right-hand side.

Keywords. sparse matrix algorithms, Gaussian elimination, partial pivoting, graph algorithms.

Subject Classification. AMS/MOS: 65F05, 65F50, 68R10. CR: F.2.1, G.1.3.

* Research partially supported by the National Science Foundation under grant DCR–85–14404, and by IBM Corporation under contract 573616.

† Research partially supported by the National Science Foundation under grant DCR–84–51385. Some of this work was done while this author was a visiting professor at the University of Iceland.

‡ Research partially supported by the National Science Foundation under a graduate fellowship.

1. Introduction.

Translating a matrix algorithm from a dense setting to a sparse setting may involve more than just generalizing indices and using lists instead of dense arrays. For example, a column-oriented Gaussian elimination algorithm computes the inner product of two vectors for each element of the matrix. If the vectors are sparse, some of the component multiplications are unnecessary. However, since it may be necessary to pivot based on previously updated elements, a sparse Gaussian elimination algorithm cannot know the exact nonzero structure of these vectors in advance of all numerical computation. This paper addresses the problem of predicting enough of the nonzero structure in advance to avoid any unnecessary computation.

For Cholesky factorization (that is, Gaussian elimination on a symmetric positive definite matrix with no pivoting), there are techniques to avoid unnecessary computation that use a graph-theoretic characterization of the nonzero structure of the Cholesky factor in terms of the original matrix [9]. These techniques break the computation into two stages. The first stage predicts the nonzero structure of the factor, which takes time proportional to its size. The second stage uses this structure in a clever way to avoid unnecessary arithmetic or data manipulation; see George and Liu [4] for a full description.

This two-stage approach is not possible for general Gaussian elimination, because pivoting makes it impossible to predict the nonzero structure of the factors from that of the matrix. The solution we propose is to break the computation of each column of the factors into a symbolic and a numeric stage. That is, each column of the factorization is computed by first predicting its nonzero structure, and then using this information to limit the numerical factorization to necessary arithmetic. Note that the number of operations used to predict the structure cannot be allowed to dominate the number of arithmetic operations.

Gaussian elimination performs exactly the same element multiplications as does matrix multiplication, except that divisions replace the multiplications by diagonal elements of the upper triangular factor. This can be seen by rewriting the decomposition $A = LU$, with L unit lower triangular and U upper triangular, to give expressions for the elements of L and U:

$$u_{ij} = a_{ij} - \sum_{k=1}^{j-1} l_{ik}u_{kj}$$

and

$$l_{ij} = \left(a_{ij} - \sum_{k=1}^{j-1} l_{ik}u_{kj}\right) \Big/ u_{jj}.$$

The elements of L and U can be determined in column major order using the equations above. If L and U are dense, this requires $2n^3/3+O(n^2)$ multiplications.[1] If they are sparse, some of these multiplications are unnecessary because at least one of the multipliers is zero. Our goal in this paper is to develop an algorithm whose total running time, including manipulation of data structures, is proportional to the number of nonzero multiplications, for every nonsingular matrix A.

[1] We use the following asymptotic notation: $f(n) = O(g(n))$ means there are constants c and m such that $n > m$ implies $|f(n)| \leq |cg(n)|$; $f(n) = o(g(n))$ means for all $c > 0$ there exists an m such that $n > m$ implies $|f(n)| \leq |cg(n)|$; and $f(n) = \Theta(g(n))$ means $f(n) = O(g(n))$ and $g(n) = O(f(n))$.

We are only interested in getting an optimal algorithm to perform Gaussian elimination, not an optimal algorithm to find an LU factorization. It is possible to perform dense LU factorization in $o(n^3)$ time by using the fast recursive matrix multiplication algorithms of Strassen and others [2, 11], but such algorithms seem to be practical only for very large dense problems (if at all).

We present the theoretical justification and development of the algorithm in section 2. In section 3, we discuss some of the details of the implementation. In section 4, we compare our implementation with some well-known sparse partial pivoting codes. We conclude by summarizing the key ideas of this paper and proposing some improvements and open questions for future research.

2. Theoretical results.

Terminology. For matrices or vectors X and Y, define flops(XY) to be the number of multiplications of nonzero elements performed while computing the product XY by conventional matrix multiplication. Gaussian elimination uses $\Theta(\text{flops}(LU))$ nonzero arithmetic operations to factor A as L times U, and our goal in this section is to develop an algorithm for Gaussian elimination (with partial pivoting) that requires total time $O(\text{flops}(LU))$ to factor an arbitrary nonsingular matrix A as $PA = LU$.

The parameters of the time analysis are as follows: n is the dimension of the matrix A; m is the number of nonzeros in A; m^* is the number of nonzeros in the factorization (formally, in $L - I + U$); flops(LU) is the number of nonzero multiplications performed. For any nonsingular A,

$$n \leq m \leq m^* \leq n^2 \qquad \text{and} \qquad m^* \leq \text{flops}(LU) < n^3.$$

For dense matrices, $m = m^* = n^2$ and flops$(LU) = \Theta(n^3)$; for sparse matrices, a typical case is a two-dimensional mesh problem with $m = \Theta(n)$, $m^* = \Theta(n \log n)$, and flops$(LU) = \Theta(n^{3/2})$ [8]. Notice that flops(LU) is likely to be $o(n^2)$, so our algorithm must not spend as much as $\Theta(n)$ time per column manipulating sparse data structures.

The algorithm. The basis for our algorithm is column-oriented LU factorization. We use the following notation: j is the index of the column of L and U being computed. Unprimed variables mean the part of a matrix or vector above row j; primed variables mean the part at or below row j. Thus $a_j = (a_{1j}, \ldots, a_{j-1,j})^T$, $a'_j = (a_{jj}, \ldots, a_{nj})^T$,

$$L_j = \begin{pmatrix} l_{11} & \cdots & 0 \\ \vdots & \ddots & \vdots \\ l_{j-1,1} & \cdots & l_{j-1,j-1} \end{pmatrix}, \qquad L'_j = \begin{pmatrix} l_{j1} & \cdots & l_{j,j-1} \\ \vdots & \ddots & \vdots \\ l_{n1} & \cdots & l_{n,j-1} \end{pmatrix},$$

$l_j = (l_{jj}, \ldots, l_{nj})^T$, and $u_j = (u_{1j}, \ldots, u_{j-1,j})^T$. (Note that $l_{jj} = 1$.) Also we use $b_j = (b_{jj}, \ldots, b_{nj})^T$ as an intermediate result. Figure 1 is a sketch of column j of L and U overwriting column j of A.

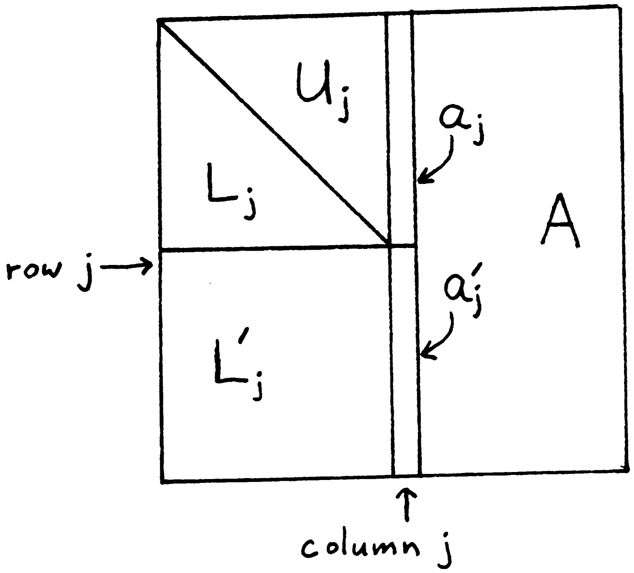

Figure 1. Computing column j of L and U.

```
0.  { Column-oriented $LU$ factorization: }
1.  for $j := 1$ to $n$ do
2.      { Compute column $j$ of $U$ and $L$ as follows. }
3.      Solve $L_j u_j = a_j$ for $u_j$;
4.      $b_j := a'_j - L'_j u_j$;
5.      Pivot: Swap $b_{jj}$ with the largest-magnitude element of $b_j$;
6.      $u_{jj} := b_{jj}$;
7.      $l_j := b_j / u_{jj}$
8.      od
```

Although partial pivoting was the motivation for all of this work, the pivoting itself is the simplest part of the algorithm.

We shall represent a column vector as a linked list of records, each containing a value and a row index. The row indices need not be in increasing order. We shall represent a matrix as an array of column vectors indexed from 1 to n.

Lemma 1. *The triangular system in step 3 of the algorithm can be solved in* $O(\text{flops}(L_j u_j))$ *time on a RAM.*

Proof. Observe that $\text{flops}(L_j u_j)$ is the total number of nonzeros of L in columns that correspond to nonzeros of u_j. The algorithm has two steps: It first determines the positions of the nonzeros in u_j and the order in which to solve for them, and then does the numeric computation.

Let $G = G(L_j)$ be the directed graph that has $j-1$ vertices, with an edge from vertex k

to vertex i iff $l_{ik} \neq 0$. Consider the nonzero positions of a_j as a set of vertices of G, and the nonzero positions of u_j as another set of vertices. By Theorem 5.1 of Gilbert [7], the nonzero positions of u_j are the vertices reachable from the vertices of a_j by paths in G. (The graph in that theorem is actually the reverse of G.) We can find those vertices by doing a depth-first search from the vertices of a_j. The data structure for L_j is an adjacency-list representation of G, so the search takes time proportional to the number of edges traversed, which is $O(\text{flops}(L_j u_j))$.

Normally, we would solve for the unknowns in increasing order of row number. The depth-first search may not find the nonzero positions of u_j in this order, however, and there is not time to sort the positions into order. (Even a bucket sort would take $\Theta(n)$ time, which would be $\Theta(n^2)$ for the whole factorization.) However, since row i of L_j has nonzeros only in those columns k for which (k, i) is an edge of G, it is possible to solve for u_{ij} as soon as all such u_{kj} have been computed. That is, the values of u_j can be solved in any topological order; increasing row order is just one possible topological order. An easy topological order to find during the depth-first search is reverse postorder [1, Problem 5.7(d)]: The linked list representing column vector u_j is initially empty. Each time the depth-first search backtracks from a vertex, that vertex is added to the head of the list. When the search is complete, the list consists of the nonzero positions of u_j in topological order.

Computing the values of u_j, then, is just a standard column-oriented lower triangular solve, in the topological order determined in the previous step:

```
1.  Copy $a_j$ into a dense vector to represent $u_j$;
2.  for each $k$ with $u_{kj} \neq 0$ (in topological order) do
3.      $u_j := u_j - u_{kj}(l_{1k}, \ldots, l_{j-1,k})^T$;
4.      od;
5.  Copy the dense vector $u_j$ into the sparse vector $u_j$
```

We compute u_j as a dense n-vector, so that in step 3 we can subtract a multiple of column k of L_j from it in constant time per nonzero in that column. Since we know the nonzero structure of u_j before we start, we need only initialize and manipulate the positions in this dense vector that correspond to nonzero positions; thus the whole thing still takes only $O(\text{flops}(L_j u_j))$ time.

The one remaining issue is that the depth-first search must mark the vertices it has reached, to avoid repeating parts of the search. These marks are normally kept in an array indexed from 1 to n, but if $\text{flops}(L_j u_j) < n$ there is not time to initialize this array to "unmarked." We could avoid initializing the array by a trick spelled out in Problem 2.12 of Aho, Hopcroft, and Ullman [1]. For our purposes, however, it suffices to initialize the array once at the beginning of the whole LU factorization, and then unmark only the marked vertices at the end of each column computation. ∎

Lemma 2. *The computation of b_j in step 4 of the algorithm can be done in $O(\text{flops}(L'_j u_j))$ time on a RAM.*

Proof. This looks just like the second part of the computation of u_j above: Copy a'_j into a dense vector to represent b_j, then subtract an appropriate multiple of a column of L'_j from it for each nonzero in u_j. ∎

Theorem 1. *The entire algorithm for LU factorization with partial pivoting can be implemented to run in $O(\text{flops}(LU))$ time on a RAM.*

Proof. By Lemmas 1 and 2, steps 3 and 4 take total time $O(\text{flops}(LU))$ plus $O(n)$ to initialize the mark array for the depth-first search. Steps 5 and 7 each examine every nonzero in L once, so they take time $O(m^*)$ overall. Step 6 takes $O(n)$ time overall, so the total is $O(\text{flops}(LU))$. ∎

3. Implementation details. We have implemented the partial pivoting algorithm in Fortran and in Lisp. This section describes the Fortran implementation, and section 4 compares that implementation with several Fortran partial pivoting codes in the literature. The Lisp implementation will be described elsewhere.

The call to the subroutine is

```
call lufact (pivot, n, a, arow, acolst, maxlu, lastlu, lu, lurow,
        lcolst, ucolst, perm, dense, found, parent, child, error).
```

The boolean argument **pivot** is set **.true.** to perform partial pivoting and **.false.** to perform no pivoting. The integer **n** is the dimension of the matrix. The arrays **a**, **arow**, and **acolst** are the matrix to be factored, in the format described in the next paragraph. The variable **maxlu** is the maximum number of nonzeros allowed in $L-I+U$, which is the size of arrays **lu** and **lurow**. The subroutine returns P, L, and U such that $PA = LU$ as follows: The integer **lastlu** is m^*, the number of nonzeros in $L - I + U$. The arrays **lu**, **lurow**, **lcolst**, and **ucolst** are L and U in the format described in the next paragraph. The array **perm** is the pivoting matrix P represented as an array of n integers; if **perm**$(r) = s$ then row r of A is in position s of PA. The implementation sets **error** to 0 to indicate success, 1 if a pivot was zero, and 2 if there was not enough space. The arrays **dense**, **found**, **parent**, and **child** are used for working storage.

Sparse column vectors are represented not as linked lists but as paired arrays of values and row subscripts, similar to the data structures used in Sparspak [4]. Thus **a(1)** through **a(**m**)** are the nonzero values of A with the nonzeros in each column contiguous but not necessarily in increasing row order; **arow(**i**)** is the row index of the nonzero **a(**i**)**; **acolst(**j**)** is the index in **a** and **arow** of the first nonzero in column j; and **acolst(**$n+1$**)** is $m+1$, the index one past the last element in **a** and **arow**. The columns of $L - I$ and U are interspersed, so **lu** contains all the nonzeros in $L - I + U$, with **lurow** giving the row indices; **lcolst** and **ucolst** are the indices of the beginnings of columns of L and U respectively.

The blocks L_j and L'_j of L above and below the current row are not stored separately. The depth-first search in Lemma 1 examines entire columns of L (instead of just columns of L_j), but it only continues the search from nonzeros in rows lower than j. This does not increase the asymptotic complexity of the algorithm. Storage in the sparse data structure for u_j is allocated (in topological order) during the depth-first search. Then storage is allocated for the nonzeros of l_j that are nonzero in A. Then the numeric computations of u_j and b_j are done together, traversing an entire column of L for each nonzero of u_j. When a nonzero of b_j is found that was zero in A, storage is allocated for the corresponding nonzero of l_j.

When the routine copies a completed column from the dense vector into the sparse data structure, it checks for exact zeros (which arise from "lucky" cancellation that could not have been predicted from the structure of the problem) and does not copy them. Our experience is that this test adds only a few percent to the running time (formally, it adds $O(m^*)$ time, which is not the leading term) on most problems, but it saves a great deal of time and storage on a few problems that have extensive "lucky" cancellation. The final factorization, therefore, stores no zero values at all.

When rows are exchanged during a pivot step, the only actual data movement is to swap one value from l_j to u_j and to record the swap in `perm`. In fact, the routine keeps the rows numbered according to their original position in A until the very end, so it actually computes P, $P^T L$, and $P^T U$; after the factorization, it uses $O(m^*)$ time to renumber so that it returns P, L, and U.

For testing our implementation, we coded routines `lsolve` and `usolve` to solve $Ly = Pb$ and $Ux = y$. They run in total time $O(m^*)$—that is, as usual, they use each nonzero element of L and U once. It would be possible to use the ideas of Lemma 1 to make these routines run in time $O(\text{flops}(Ly))$ and $O(\text{flops}(Ux))$ respectively, but we did not do so.

All working arrays are passed to `lufact` by the user; the routine does not declare any local array storage. The sizes of the arrays used are as follows.

`a`	m	reals
`arow`	m	integers
`acolst, ucolst`	$n+1$	integers
`lu`	m^*	reals (maximum is `maxlu`, actual is `lastlu`)
`lurow`	m^*	integers
`lcolst, perm, parent, child`	n	integers
`dense`	n	reals
`found`	n	booleans

If the same array is passed to `a` and `lu`, then `lu` and `lurow` will overwrite `a` and `arow`. In this case the total array storage is $m^* + n$ reals and $m^* + 7n$ integers and booleans.

4. Comparison with other algorithms. Some of the existing algorithms to solve sparse linear systems, although in theory dominated by data manipulation costs, have implementations that display very efficient behavior on real-world problems. For comparison with our implementation, we chose Iain Duff's MA28 [3], Andrew Sherman's NSPIV [10], and the code of Alan George and Esmond Ng [5, 6], referred to here as NG.

The NG code uses a heuristic to predict the nonzero structures of the factors before doing any numerical factorization. It pre-allocates storage for every position that could possibly be nonzero in some pivotal sequence. No dynamic storage allocation is necessary during numeric factorization, but some unnecessary storage may be pre-allocated. A factorization can be re-used with different right-hand sides. Pre-allocated storage can be re-used by a sequence of matrices with identical structure. The columns of the original matrix must be ordered to reduce fill in the predicted structure for this method to effectively exploit sparsity. The authors recommend a minimum degree order.

The NSPIV code solves by row using column partial pivoting, obtaining the factorization $AQ = LU$. It makes updates on the basis of the following rows, and allocates storage for fill dynamically. It does not save the lower triangular factor, so that a factorization cannot be re-used with a different right-hand side. Since there is no pivoting for sparsity, the user must supply appropriate row and column orders for the matrix.

The MA28 code does row and column permutations to maintain numerical stability and preserve sparsity simultaneously. The user sets a threshold value for pivoting to balance the stability and sparsity considerations. Block lower triangular pre-ordering is available (and recommended) since only the diagonal blocks need to be factored. A factorization can be re-used with different right-hand sides. A matrix that is identical in structure and pivot sequence to a previously factored matrix can use the information from the latter to allocate the required storage in advance and avoid some unnecesary arithmetic.

Our test problems, described in Table 1, were nine finite element problems and seven other problems. We pre-ordered the columns of each matrix by minimum degree. For the finite element problems, we chose random numerical values. In all cases, we chose the right-hand side to make the correct solution be $(1, 1, \ldots, 1)^T$. We compared the codes on a Symbolics 3675 Lisp Machine with a floating point accelerator, using the Symbolics Fortran 77 compiler, with single precision arithmetic.

For these tests, we selected a representative sample of sparse matrix problems and codes to determine whether our theoretical expectations would be realized in practice. We do not claim to have made a comprehensive survey of sparse Gaussian elimination codes, nor do we claim to have represented all types of real-world problems in our experiments.

Table 2 gives the results. The column headed M^* is the number of nonzeros in the factors (in U alone, for NSPIV). Column T is the time in seconds for factorization plus one forward- and back-solve. (A call to NSPIV always includes a solution; in the other codes, the solution time is invariably only a small fraction of the total.) The next two columns measure space: Column $R + I$ is the total array space used, assuming a real number and an integer are the same size (which was the case for these experiments), and column $2R+I$ is array space, assuming a real number to be twice as large as an integer (which would be the case for double-precision versions of the codes). The final column is the error in the computed solution, in the infinity norm.

Table 3 gives the same results as Table 2, with the results for each problem normalized so that the best result is 1. Table 4 gives averages of these normalized results (with all 16 problems weighted equally). We omitted NSPIV from the normalized results because it only stores the upper triangular factor.

We now discuss the results, considering each code in turn.

Since MA28 uses different pivoting criteria than the other codes, we give separate results for each of two settings of the threshold parameter u. The setting $u = 0.1$ is meant to reflect the "well-tuned" behavior of MA28. The setting $u = 1.0$ is meant to be an approximation to partial pivoting, although MA28 will still break ties for sparsity. The storage we report for MA28 is the minimum storage needed to solve the problem. In fact, we granted the routine some extra "elbow room" to allow it to run efficiently. Duff [3] and George and Ng [5] discuss MA28's elbow room requirements. The new code was usually substantially faster than either version of MA28, and used about the same amount of array storage.

Since NSPIV pivots on columns rather than rows, we pre-ordered the columns of A by minimum degree, and then factored A^T. This presents NSPIV with the same pivoting choices at each major step as those faced by NG and the new code. (Incidentally, this made NSPIV perform much better than it did when factoring $\boldsymbol{A}$ with no row pre-ordering.) It is difficult to compare NSPIV directly with the other codes. Because it discards $\boldsymbol{L}$, it saves considerable storage and some data manipulation time, and the factorization can only be used once. On most of the problems, NSPIV used less storage and somewhat less time than the new code. Discarding $\boldsymbol{L}$ probably accounts for all of the storage advantage; it is not clear whether it also accounts for the time advantage.

The NG code generally produced the same actual fill as the new code, which is to be expected since both pivot within columns. It saves overhead storage by a subscript compression technique, but uses extra storage because it can overestimate potential fill. In the end, its total storage requirement was very close to that of the new code; usually NG was better in single precision and the new code was better in double precision. The new code generally ran somewhat faster.

5. Conclusions and future work. The theoretical contribution of this paper is an algorithm to perform sparse $\boldsymbol{LU}$ factorization in which data structure manipulation does not dominate necessary arithmetic, even in the worst case. The key idea is a lower triangular solver that uses depth-first search and topological ordering to take advantage of sparsity in the right hand side. Implementation of this idea requires only simple data structures, and it gives a code that is competitive with and often superior to existing sparse $\boldsymbol{LU}$ codes.

Avoiding $O(n)$ overhead time per column is not merely a theoretical nicety. When we added a debugging loop to check after each column that every element of the dense vector was set to zero, running time increased enormously.

It appears that data manipulation does often dominate the running times of existing algorithms. By forcing the total number of operations to be proportional to the number of nonzero flops, we achieved running times that were faster, on average, than codes without this constraint.

Several possible improvements could be made to this code. We mention some of them here.

The code's overhead storage requirement is dominated by the row numbers of the nonzeros in $\boldsymbol{L}$ and $\boldsymbol{U}$. Ng's code [6] reduces this requirement, often by more than half, by using a compressed data structure to overlap the row numbers of consecutive columns with similar nonzero patterns. This would not work in our code because the columns are not kept in increasing row order, and sorting them one column at a time would be too slow. We could build up the columns, unsorted, in an uncompressed data structure, and then sort them into a compressed structure efficiently at the end; however, the intermediate storage would be just as large as at present. A compromise approach is to build up unsorted and uncompressed columns, stopping periodically to sort and compress them. If we stop to sort and compress after every $\Theta(n)$ nonzeros are generated, the total time for sorting is only $O(m^*)$ (with a suitable lexicographic bucket sort [1, Problem 3.16]) and the storage is only $O(n)$ more than the fully compressed data structure. We plan to experiment with this approach in the future.

Another way to save space is to do threshold pivoting: Instead of pivoting on the element of largest magnitude in the column, choose an element on the basis of sparsity from among those elements whose magnitude is at least some threshold factor $u < 1$ times the maximum. An appropriate choice might be that element whose row contains the fewest nonzeros to the right of the active column. We plan experiments here as well.

There are also some open theoretical questions about this algorithm. Since the algorithm computes the factors column by column, it does not have available the values of the partially factored matrix to the right of the active column. Therefore, it cannot do complete pivoting (that is, pivoting on the element of largest magnitude in the remaining partially eliminated submatrix) or any kind of dynamic column pivoting for sparsity (for example, the Markowitz method of pivoting on the element that will cause the smallest update to the partially eliminated submatrix). Is there an LU factorization algorithm that runs in $O(\text{flops}(LU))$ time and makes the partially eliminated submatrix available at each step? The hard part of such an algorithm would be to compute the update $A := A - vw^T$ for sparse A, v, and w in time proportional to arithmetic operations.

Lemma 1 gives an algorithm to solve a lower (or upper) triangular system $Lx = b$ in $O(\text{flops}(Lx))$ time if L is represented by columns. Is there a similar algorithm if L is represented by rows? This seems unlikely. However, perhaps there is such an algorithm for the special case where L is the Cholesky factor of a symmetric, positive definite matrix (and therefore is the adjacency matrix of a chordal graph [9]).

Finally, there are applications that require $Ax = b$ to be solved for many different b with the same A, but that only require some of the elements of x for each b. (See Gilbert [7] for a reference to one such application.) It would be interesting to investigate the number of operations necessary in factorization and solution of such systems, and to develop sparse algorithms that run within the theoretical time bounds.

Acknowledgements. The authors wish to express their gratitude to Tom Coleman, who first pointed out that $Lu = a$ could be solved in any topological order. Tom participated in numerous discussions of the implementation and gracefully endured the long wait for experimental results. Esmond Ng very kindly provided us with his library of examples for testing and his partial pivoting codes for comparison. Much of the implementation was done while the first author was a visitor at the University of Iceland at Reykjavík; he wishes to thank the University, and Oddur Benediktsson and Sven Sigurðsson in particular, for providing a stimulating environment during the fall of 1985.

Prob	N	M	Remarks
1	113	655	Matrix pattern supplied by Morven Gentleman.
2	199	701	Matrix pattern supplied by Willoughby.
3	130	1282	Matrix from laser problem (A.R. Curtis).
4	663	1712	Basis from linear programming problem.
5	363	3279	Basis from linear programming problem.
6	822	4841	Basis from linear programming problem.
7	541	4285	Facsimile convergence matrix.
8	936	6264	Finite element mesh — a hollow square (small hole).
9	1009	6865	Finite element mesh — a graded-L.
10	1089	7361	Finite element mesh — a plain square.
11	1440	9504	Finite element mesh — a hollow square (large hole).
12	1180	7750	Finite element mesh — a +-shaped domain.
13	1377	8993	Finite element mesh — an H-shaped domain.
14	1138	7450	Finite element mesh — three holes.
15	1141	7465	Finite element mesh — six holes.
16	1349	9101	Finite element mesh — a pinched hole.

Table 1.

Prob	N	M	Code	M*	T	R+I	2R+I	Error
1	113	655	New	1349	0.52	3604	5066	2.43−4
			NG	1349	0.58	3859	5986	1.04−4
			NSPIV	348	0.23	3026	4368	3.37−4
			MA28$_{u=0.1}$	698	0.62	4018	5078	2.78−4
			MA28$_{u=1.0}$	861	1.05	4356	5585	2.56−4
2	199	701	New	2432	0.87	6458	9089	2.18−4
			NG	2433	1.17	7415	11612	6.64−4
			NSPIV	960	0.58	5116	7374	1.00−2
			MA28$_{u=0.1}$	1443	1.20	6684	8581	1.42−3
			MA28$_{u=1.0}$	2053	3.40	7936	10459	1.17−3
3	130	1282	New	9158	12.02	19358	28646	7.42−2
			NG	9158	12.93	17510	33195	8.59−2
			NSPIV	7402	21.17	18541	27615	6.09−3
			MA28$_{u=0.1}$	1180	1.18	6064	7740	7.81−2
			MA28$_{u=1.0}$	1180	1.18	6064	7740	7.81−2
4	663	1712	New	2263	0.65	9832	12758	0.00+0
			NG	2090	1.63	23857	36438	0.00+0
			NSPIV	612	0.29	10618	14931	0.00+0
			MA28$_{u=0.1}$	663	0.75	15081	18119	0.00+0
			MA28$_{u=1.0}$	663	0.77	15081	18119	0.00+0
5	363	3279	New	6370	2.83	15646	22379	1.17−3
			NG	6370	3.60	19149	30492	9.21−4
			NSPIV	2217	1.97	14262	20847	5.71−4
			MA28$_{u=0.1}$	3012	4.13	17616	22788	1.60−4
			MA28$_{u=1.0}$	3311	5.20	18200	23664	1.11−4
6	822	4841	New	17546	8.07	41670	60038	4.09−3
			NG	17557	27.16	96461	169879	5.71−3
			NSPIV	8568	10.13	34219	50094	2.39−4
			MA28$_{u=0.1}$	3282	3.78	28641	36020	5.78−4
			MA28$_{u=1.0}$	3539	4.36	29177	36824	3.68−4
7	541	4285	New	15337	7.79	35004	50882	3.41−2
			NG	15331	11.48	40618	67298	3.41−2
			NSPIV	6989	6.04	27420	40317	6.13−1
			MA28$_{u=0.1}$	15439	36.89	47212	64385	5.94−2
			MA28$_{u=1.0}$	13734	34.75	42699	58149	5.66−2
8	936	6264	New	46695	42.73	100880	148511	3.98−2
			NG	46652	54.06	84138	147638	4.26−2
			NSPIV	19807	33.90	60569	89448	1.54−2
			MA28$_{u=0.1}$	41646	250.79	111687	155463	1.31−1
			MA28$_{u=1.0}$	50000	484.63	132443	184565	7.84−3

Table 2. (Problems 1–8)

Prob	N	M	Code	M*	T	R+I	2R+I	Error
9	1009	6865	New	64724	79.47	137522	203255	2.68−3
			NG	64724	92.47	110600	197723	2.15−2
			NSPIV	27924	64.23	78662	116478	5.41−3
			$\text{MA28}_{u=0.1}$	52307	282.95	140688	195290	1.19+1
			$\text{MA28}_{u=1.0}$	68891	1022.70	184034	255262	1.08−2
10	1089	7361	New	58285	58.73	125284	184658	8.24−5
			NG	58284	74.03	104377	184967	9.41−5
			NSPIV	25490	47.05	75506	111624	1.21−4
			$\text{MA28}_{u=0.1}$	55718	324.02	149199	207441	2.28−2
			$\text{MA28}_{u=1.0}$	65780	813.83	174756	243075	8.92−5
11	1440	9504	New	60035	43.23	131592	193067	6.65−5
			NG	60027	55.23	114594	198203	2.19−4
			NSPIV	25383	33.91	82737	121944	5.83−4
			$\text{MA28}_{u=0.1}$	54250	374.95	147089	204447	6.24−2
			$\text{MA28}_{u=1.0}$	62790	730.37	169459	235388	2.81−4
12	1180	7750	New	33533	16.55	76508	111221	5.64−5
			NG	33534	22.22	73320	122660	1.50−4
			NSPIV	14454	12.30	55031	80775	7.96−5
			$\text{MA28}_{u=0.1}$	33983	105.28	98874	135362	1.44−2
			$\text{MA28}_{u=1.0}$	38349	200.88	109729	150632	7.36−5
13	1377	8993	New	35549	15.97	82116	119042	1.89−4
			NG	35555	24.18	79167	130954	1.53−4
			NSPIV	14932	11.65	60246	88302	2.95−4
			$\text{MA28}_{u=0.1}$	34892	90.15	104313	142098	9.58−2
			$\text{MA28}_{u=1.0}$	40641	244.70	117716	161253	1.17−4
14	1138	7450	New	44480	30.97	98066	143684	1.47−4
			NG	44481	40.61	85879	147213	5.99−5
			NSPIV	18607	24.27	62359	91830	5.87−5
			$\text{MA28}_{u=0.1}$	41621	273.82	113511	157657	1.35−2
			$\text{MA28}_{u=1.0}$	48976	395.09	130842	182381	5.15−5
15	1141	7465	New	47249	37.15	103628	152018	2.09−4
			NG	47246	47.02	90550	155473	1.32−4
			NSPIV	20075	29.35	65352	96315	6.88−5
			$\text{MA28}_{u=0.1}$	45143	292.45	122223	169877	5.60−1
			$\text{MA28}_{u=1.0}$	53859	476.95	142162	198611	1.65−4
16	1349	9101	New	77853	82.92	166500	245702	9.66−5
			NG	77852	103.56	136734	243843	1.02−4
			NSPIV	33302	66.82	96950	143400	4.77−5
			$\text{MA28}_{u=0.1}$	71135	786.78	187345	261495	2.08−2
			$\text{MA28}_{u=1.0}$	92049	1719.43	238323	333481	1.17−4

Table 2. (Problems 9–16)

Prob	N	M	Code	M*	T	R+I	2R+I
1	113	655	New	1.93	1.00	1.00	1.00
			NG	1.93	1.12	1.07	1.18
			MA28$_{u=0.1}$	1.00	1.19	1.11	1.00
			MA28$_{u=1.0}$	1.23	2.02	1.21	1.10
2	199	701	New	1.69	1.00	1.00	1.06
			NG	1.69	1.34	1.15	1.35
			MA28$_{u=0.1}$	1.00	1.38	1.03	1.00
			MA28$_{u=1.0}$	1.42	3.91	1.23	1.22
3	130	1282	New	7.76	10.19	3.19	3.70
			NG	7.76	10.96	2.89	4.29
			MA28$_{u=0.1}$	1.00	1.00	1.00	1.00
			MA28$_{u=1.0}$	1.00	1.00	1.00	1.00
4	663	1712	New	3.41	1.00	1.00	1.00
			NG	3.15	2.51	2.43	2.86
			MA28$_{u=0.1}$	1.00	1.15	1.53	1.42
			MA28$_{u=1.0}$	1.00	1.18	1.53	1.42
5	363	3279	New	2.11	1.00	1.00	1.00
			NG	2.11	1.27	1.22	1.36
			MA28$_{u=0.1}$	1.00	1.46	1.13	1.02
			MA28$_{u=1.0}$	1.10	1.84	1.16	1.06
6	822	4841	New	5.35	2.13	1.45	1.67
			NG	5.35	7.19	3.37	4.72
			MA28$_{u=0.1}$	1.00	1.00	1.00	1.00
			MA28$_{u=1.0}$	1.08	1.15	1.02	1.02
7	541	4285	New	1.12	1.00	1.00	1.00
			NG	1.12	1.47	1.16	1.32
			MA28$_{u=0.1}$	1.12	4.74	1.35	1.27
			MA28$_{u=1.0}$	1.00	4.46	1.22	1.14
8	936	6264	New	1.12	1.00	1.20	1.01
			NG	1.12	1.27	1.00	1.00
			MA28$_{u=0.1}$	1.00	5.87	1.33	1.05
			MA28$_{u=1.0}$	1.20	11.34	1.57	1.25

Table 3. (Problems 1–8)

Prob	N	M	Code	M*	T	R+I	2R+I
9	1009	6865	New	1.24	1.00	1.24	1.04
			NG	1.24	1.16	1.00	1.01
			MA28$_{u=0.1}$	1.00	3.56	1.27	1.00
			MA28$_{u=1.0}$	1.32	12.87	1.66	1.31
10	1089	7361	New	1.05	1.00	1.20	1.00
			NG	1.05	1.26	1.00	1.00
			MA28$_{u=0.1}$	1.00	5.52	1.43	1.12
			MA28$_{u=1.0}$	1.18	13.86	1.67	1.32
11	1440	9504	New	1.11	1.00	1.15	1.00
			NG	1.11	1.28	1.00	1.03
			MA28$_{u=0.1}$	1.00	8.67	1.28	1.06
			MA28$_{u=1.0}$	1.16	16.89	1.48	1.22
12	1180	7750	New	1.00	1.00	1.04	1.00
			NG	1.00	1.34	1.00	1.10
			MA28$_{u=0.1}$	1.01	6.36	1.35	1.22
			MA28$_{u=1.0}$	1.14	12.14	1.50	1.35
13	1377	8993	New	1.02	1.00	1.04	1.00
			NG	1.02	1.51	1.00	1.10
			MA28$_{u=0.1}$	1.00	5.64	1.32	1.19
			MA28$_{u=1.0}$	1.16	15.32	1.49	1.35
14	1138	7450	New	1.07	1.00	1.14	1.00
			NG	1.07	1.31	1.00	1.02
			MA28$_{u=0.1}$	1.00	8.84	1.32	1.10
			MA28$_{u=1.0}$	1.18	12.76	1.52	1.27
15	1141	7465	New	1.05	1.00	1.14	1.00
			NG	1.05	1.27	1.00	1.02
			MA28$_{u=0.1}$	1.00	7.87	1.35	1.12
			MA28$_{u=1.0}$	1.19	12.84	1.57	1.31
16	1349	9101	New	1.09	1.00	1.22	1.01
			NG	1.09	1.25	1.00	1.00
			MA28$_{u=0.1}$	1.00	9.49	1.37	1.07
			MA28$_{u=1.0}$	1.29	20.74	1.74	1.37

Table 3. (Problems 9–16)

Code	M*	T	R+I	2R+I
New	2.07	1.65	1.25	1.22
NG	2.05	2.34	1.39	1.65
$\mathrm{MA28}_{u=0.1}$	1.01	4.61	1.26	1.10
$\mathrm{MA28}_{u=1.0}$	1.17	9.02	1.41	1.23

Table 4.

References.

[1] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, 1975.

[2] D. Coppersmith and S. Winograd. On the asymptotic complexity of matrix multiplication. *SIAM Journal on Computing* **11**: 472–492, 1982.

[3] I. S. Duff and J. K. Reid. Some design features of a sparse matrix code. *ACM Transactions on Mathematical Software* **5**: 18–35, 1979.

[4] Alan George and Joseph W-H Liu. *Computer Solution of Large Sparse Positive Definite Systems.* Prentice-Hall, 1981.

[5] Alan George and Esmond Ng. An implementation of Gaussian elimination with partial pivoting for sparse systems. *SIAM Journal on Scientific and Statistical Computing* **6**: 390–409, 1985.

[6] Alan George and Esmond Ng. Symbolic factorization for sparse Gaussian elimination with partial pivoting. University of Waterloo report CS–84–43, 1984.

[7] John R. Gilbert. Predicting structure in sparse matrix computations. Cornell University report CS–86–750, 1986.

[8] John R. Gilbert and Robert Schreiber. Nested dissection with partial pivoting. Sparse Matrix Symposium, Fairfield Glade, Tennessee, 1982.

[9] Donald J. Rose. A graph-theoretic study of the numerical solution of sparse positive definite systems of linear equations. In Ronald C. Read (editor), *Graph Theory and Computing*, pages 183–217, Academic Press, 1972.

[10] Andrew H. Sherman. Algorithm 533. NSPIV, a FORTRAN subroutine for sparse Gaussian elimination with partial pivoting. *ACM Transactions on Mathematical Software* **4**: 391–398, 1978.

[11] V. Strassen. Gaussian elimination is not optimal. *Numerische Mathematik* **13**: 354–356, 1969.