High-Performance Out-of-Core Sparse LU Factorization*

John R. Gilbert† Sivan Toledo‡

Abstract

We present an out-of-core sparse nonsymmetric LU-factorization algorithm with partial pivoting. We have implemented the algorithm and our experiments show that it can easily factor matrices whose factors are larger than main memory at rates comparable to those of an in-core solver. The algorithm is novel in several respects, including the use of panels that are larger than memory and the use of a priority queue of updates.

1 Introduction.

We present an algorithm for out-of-core sparse LU factorization with partial pivoting. A user may fail to solve a large linear system because a solver breaks down numerically, runs for too long, or runs out of memory. Although most of the research in high-performance scientific computing has focused on reducing running times by exploiting parallelism and locality, many users' failure to solve large systems stems from running out of memory. Our algorithm allows users to factor matrices that are larger than main memory and whose factors are larger than main memory, at rates similar to those of in-core solvers.

Preconditioned iterative solvers, using either incomplete factorization preconditioners or approximate inverse preconditioners, have been also been proposed as a way to solve large linear systems whose LU factors do not fit within main memory. Preconditioned iterative solvers, however, may break down numerically or fail to converge when forced to use only a small main memory. We therefore believe that an out-of-core direct solver represents one of the best approaches to the "black-box" solution of large sparse linear systems.

The amount of main memory that our algorithm uses is only proportional to the number m of rows plus the number n of columns in the matrix A. We have implemented the algorithm as a Matlab module in C. Our experience so far with the algorithm, on a Sun Ultra workstation and on an SGI Origin 200, using standard large test matrices, has been that the factorization is CPU bound and comparable in performance to Matlab's sparse LU factorization. Therefore, the new algorithm allows even naive users to solve linear systems of almost unlimited size efficiently on computers with modest memory sizes.

Section 2 describes the algorithm. The amount of non-floating-point work that the algorithm performs is analyzed in Section 3. Section 4 compares the design of our algorithm to the design of a state-of-the-art in-core LU-factorization algorithm. Our implementation

*This research was supported in part by DARPA contract number DABT63-95-C-0087 and by NSF contract number ASC-96-26298. This research was conducted while Sivan Toledo was with the Xerox Palo Alto Research Center.

†Xerox Palo Alto Research Center, 3333 Coyote Hill Road, Palo Alto, CA 94304. email: gilbert@parc.xerox.com.

‡Computer Science Department, Tel Aviv University, Tel Aviv 69978, ISRAEL. email: sivan@math.tau.ac.il.

is described in Section 5, and the results of our experiments are described in Section 6. We present our conclusions in Section 7.

2 The Algorithm

The algorithm is a block-column left-looking factorization, but it differs substantially from other block-column left-looking sparse algorithms. A block-column left-looking algorithm loads a panel of columns of A into memory, updates these columns using the columns of L to their left (smaller index), factors the panel, and appends its columns to L and U. It then loads the next panel of columns to the right, updates it, factors it, and so on. In terms of data locality and data reuse, wider panels typically yield better data reuse but require more fast memory.

The factorization algorithm consists of three main phases: symbolic analysis, scheduling, and numerical factorization. All three phases work well with a limited amount of main memory. The symbolic analysis phase computes the column elimination tree (c-etree) of the matrix, a postordering of this tree, and upper bounds on the number of nonzeros in each column of the factors L and U. For more information on the algorithms that are used in the symbolic analysis phase, see [4, 5]. The scheduling phase uses this information to compute an efficient out-of-core schedule for the factorization. The numerical factorization phase determines which column of L updates which column of A and computes the columns of U and L. As in other pivoting LU factorization algorithms, the symbolic work to determine which columns of L update each column is interleaved with numerical computations.

A matrix to be factored should normally be reordered to minimize fill. We currently rely on in-core reordering algorithms, such as colamd [1]. Relying on an in-core reordering algorithm means that matrices that fit within main memory can usually be reordered and factored efficiently, even if their factors do not fit within memory, but that matrices that are larger than memory might cause significant paging during the reordering phase (even though the factorization itself is insensitive to whether the matrix is larger than memory).

The (reordered) matrix is initially stored on disk in compressed column format. The algorithm stores the factors in the same format. The structure of the data files is described in the next section.

The following subsections describe the three phases of the algorithm in detail.

2.1 Computing the Column Elimination Tree and Bounding Column Counts

Our code computes the column elimination tree of a matrix with one pass over the matrix. This part of the algorithm uses $\Theta(m + n)$ words of memory[1], so it can be performed in core with no I/O except for reading the matrix once. Older codes scan the matrix twice to compute the column elimination tree. Since reading the matrix from disk can be expensive, we have fused the two passes into one, but the algorithm is essentially the same.

During the computation of the column elimination tree we perform one more task. We create the nonzero structure of another matrix S_A that is later used in the computation of column counts. For each nonzero (i, j) in A, S_A has a nonzero (j, f), where f is the index of the first column in A that has a nonzero in row i. The code writes the indices (j, f) to disk using a large buffer, so that actual I/O is performed infrequently when the buffer fills.

[1] We use $\Theta(n)$ to denote functions $f(n)$ whose growth is proportional to n.

Next, the algorithm computes a postorder of the column elimination tree using a depth-first traversal. This computation is performed in core using $\Theta(m)$ memory.

We now have the column elimination tree and a postorder of it in memory, and the structure of S_A is stored on a file. The structure of S_A is stored by row, possibly with duplicates. Subsequent parts of the algorithm need to access the structure of S_A by column in postorder, and without duplicates, so an out-of-core sorting algorithm sorts the row-column index pairs in the file lexicographically by column (in postorder) and by row. This enables removal of duplicates and efficient access by column.

The out-of-core sorting algorithm is a k-way merge sort. In each iteration, the algorithm reads k sorted runs and merges them together into a single sorted run. The larger the value of k, the fewer the number of iterations. Each sorted run is read sequentially using a buffer. The value of k is chosen so that there is enough main memory for k buffers, where each buffer is large enough so that I/O is performed efficiently. (Reading few large blocks from disks is more efficient than reading many small blocks). We currently use buffers of size 64Kbytes. With 64Mbytes of main memory devoted to these buffers, for example, the algorithm performs one merge to sort files of up to 64Mbytes, and two merges to sort up to 64Gbytes.

The first iteration of this sorting algorithm is actually fused with the computation of S_A. As mentioned above, we write the elements of S_A in batches using a buffer. This single buffer is as large as main memory would hold. When the buffer fills up, we sort it, then write it to disk as a sorted run. The merge-sort algorithm therefore skips the first merge iteration. Using our example of 64Mbytes of main memory, files of up to 64Gbytes are sorted in only one merge iteration.

Next, the algorithm bounds the nonzero counts for the columns of U and L. This is done by simulating an elimination process on S_A. We traverse the columns of S_A in postorder (of the c-etree of A). When the algorithm processes column j, after all of its descendants have been processed, it uses its nonzero count as the nonzero count of the jth column of L, and adds all of its nonzeros to the structure of the parent column $p(j)$ of j. The algorithm adds these nonzeros by pushing them on a stack. When the parent $p(j)$ is processed, its nonzero structure is determined by the union of the structure of column $p(j)$ in S_A with the set of nonzeros that have been pushed on the stack by the immediate children of $p(j)$. The bounds on the column counts of U are computed by keeping track of the row counts in this simulated elimination process.

Although this simulation process uses less memory than an actual elimination on S_A, we do not assume that it can be performed in core. The stack is therefore maintained out of core. It is stored in a file and accessed using a large buffer.

To summarize, the symbolic elimination phase is implemented out-of-core using three techniques. We use loop fusion to reduce the number of passes over the input matrix to one, and to fuse the first iteration of the merge sort algorithm with the generation of the matrix S_A. We use an out-of-core merge sort algorithm to transpose S_A and to remove duplicate entries. We use an out-of-core stack data structure to implement a symbolic simulation of an elimination process.

2.2 Scheduling

The next major phase in the algorithm is scheduling the numerical factorization. Before we can describe the scheduler, however, we must explain how the numerical factorization works.

The matrix is factored in panels. Each panel consists of a group of consecutive (in the postorder) columns. Conceptually, after the columns in panels 1 through $J-1$ have been factored, the algorithm loads panel J, performs all the updates from columns in previous panels to columns in panel J, and factors panel J.

Wider panels require less I/O. Consider a column j that eventually updates, say, 100 other columns to the right of it. If these 100 columns belong to 11 panels besides the panel that j belongs too, we must read j 11 times from disk. If, on the other hand, these 100 panels belong to only 3 other panels, then j is read only 3 times during the algorithm. Thus, wider panels lead to less I/O.

In fact, storing all of panel J simultaneously in memory is often unnecessary. If columns j_1 and j_2 are both in panel J, and if neither one is an ancestor of the other in the c-etree, then they do not need to be stored in memory simultaneously, because only columns that are on a single root-to-leaf path in the tree update each other. The structure of the c-etree tells us that column j_1 will not update j_2 or vice versa. The c-etree also tells us that if a column j_3 might update j_1, then it will not update j_2. Thus, not having j_1 and j_2 in memory simultaneously does not prevent any updates from occurring and does not require loading columns of L more than once per panel.

Thus, our scheduler decomposes the matrix into panels that cannot, in general, be stored in main memory. The decomposition is done, however, such that each part of a root-to-leaf path within a panel can be stored in memory. To ensure that paths in the panel can be stored in main memory, the scheduler uses the upper bounds on the column counts that were computed in the first phase of the algorithm. The scheduler decomposes the matrix using a depth-first traversal of the c-etree.

Besides decomposing the matrix into panels, the scheduler builds the complete sequence of columns to be read from disk and columns to be factored and written to disk. Since we do not have enough memory to hold an entire panel, columns within the panel must be factored and written to disk as soon as all the necessary updates have been applied to them, so that other columns in the panel can be read from disk. The schedule is broken into *steps*. In each step one or more columns of A are read from disk, updated (the updating column may also update other columns in the panel), and one or more columns are factored and written to the file containing L. The set of columns to be factored may be a subset or a superset of the columns that are loaded in the same step. The complete schedule is described by four integer vectors of size n. One vector specifies the sequence of columns to be read from disk into panels, a second the sequence of columns to be factored, a third the step in which each column is loaded into a panel, and the fourth the step in which each column is factored.

2.3 Numerical Factorization

The numerical factorization step factors A according to the schedule that was previously computed.

In the beginning of each step in the factorization, the algorithm has a set P (possibly empty) of columns in memory. The columns in memory are part of a root-to-leaf path in the c-etree. Step s starts with loading an additional set F_s of columns of A into the active set P. The augmented set P is still a partial path in the c-etree. Next, the algorithm performs several updates. These updates load columns of L from disk and use them to update columns in P. Now the algorithm factors and writes to disk a subset E_s (possibly all) of the columns of P. Each column in E_s, in postorder, is factored, used to update the

rest of the columns in P, and written to disk as part of U and L.

The sets of columns F_s and E_s for each step s are given by the schedule. The set of columns of L that are loaded from disk in step s and update P is not given by the schedule, because it depends on pivoting decisions that are made during the numerical factorization phase.

The algorithm uses a priority queue, implemented as a binary heap, to determine which columns of L must be loaded in step s. When a column of A is loaded into P and when a column of L is loaded for an update, the indices of the nonzeros that are in rows that have already been used as pivot rows are inserted into the heap. The algorithm maintains a bitmap specifying which row indices are already in the heap, to avoid duplicate indices in the heap.

Row indices in the heap are ordered according to the indices of the columns for which they served as pivots. That is, row index i will be extracted from the heap before row index i' if the pivot in column j was i and the pivot in column j' was i' and $j < j'$. During the update phase of step s, the algorithm repeatedly extracts from the heap the row i index with the smallest corresponding column index j. If j is in the panel that is currently being factored, it is skipped. (Because column j of L already updated the columns in P.) Otherwise, the algorithm loads the jth column of L from disk, uses it to update the columns of P, and adds row indices of nonzeros in column j to the heap as appropriate. That is, indices of rows that have already served as pivot rows and which are not already in the heap are added.

The heap allows the algorithm to determine that a column j of L must update some of the columns in P, but it does not tell it which ones. To determine efficiently which columns of P must be updated by a given column j of L, the algorithm also maintains the nonzero structure of P in a collection of row lists (in addition to a column-oriented representation). The structure of each row is maintained as a linked list, possibly empty. Elements are inserted into the row lists when they are loaded from disk or when fill elements are created.

The algorithm maintains the active set of columns P in one of two ways, selectable at runtime. The columns can be stored in compressed column format with space reserved for fill elements. The algorithm allocates each compressed column with enough space to store worst-case fill, as predicted by the symbolic analysis phase. Or, the columns can be stored in a sparse accumulator (spa) data structure, in which the indices of the nonzeros are stored in a compressed format, but the nonzeros themselves are stored in a full array of size m. The compressed column format allows the algorithm to hold more columns in main memory. When the columns of P are stored in compressed columns, each column must be scattered into a size-m array before a column-column update operation and gathered after the operation. The spa structure requires more space per column, so fewer columns can be stored in main memory, which leads to more I/O, but it reduces scatter-gather overhead. Which data structure is appropriate depends on whether the computation is CPU bound or I/O bound.

3 Analysis

The performance of an out-of-core algorithm may be limited either by I/O performance or by in-core computations. We have structured our algorithm to perform as little I/O as possible, and in practice it does not spend significant amounts of time waiting for I/O. We thus turn to estimating the amount of work spent in symbolic non-floating-point operations, which can consume a significant fraction of the running time.

The amount of symbolic overhead in the symbolic analysis and scheduling phases is typically insignificant. The computation of the elimination tree and its postorder are done in $O(n+m)$ time. Computing the upper bounds on the column counts of L and U cannot be bounded by a simple expression, but it is typically insignificant as well, as shown by our experimental results in Section 6. Computing the schedule requires $O(n)$ work.

Most of the symbolic overhead is incurred during the numerical factorization phase.

The amount of work to maintain the list of row indices of P is $\Theta(\text{nnz}(U+L))$, since each nonzero of A and each fill element is inserted once into the appropriate row list, visited once, and deleted. Each operation costs a constant amount of work because insertions are always done at the head of the lists[2].

Estimating the work spent on heap operations is more complex. The work required to insert or extract an element from the heap is $\Theta(\log h)$, where h is the size of the heap at the time of the operation. In our algorithm the size of the heap is bounded by the number of nonzero rows in a set P of columns. Each heap operation can be charged against the loading of one column of L from disk and performing one or more column-column updates. The cost of each column-column operation is proportional to the number of nonzeros in the column of L. Although we do not know how to theoretically relate the number and cost of heap operations to the number of nonzeros or floating point operations in the algorithm, this overhead is low. The cost of the two heap operations associated with a row index is likely to be insignificant relative to the cost of loading the corresponding column from disk and the cost of performing several column-column updates. In practice, the measured cost of heap operations is low: the number of heap operations is often more than 3 orders of magnitude less than the number of floating-point operations.

We now turn to symbolic overhead in the inner loop of the algorithm, during column-column update operations. If the columns of P are stored in compressed format, each column must be scattered to a size-m array before each column-column update and gathered back after the operation. This is a major cost. The amount of work in these scatter-gather operations cannot even be bounded by the number of floating-point operations, since a column that is scattered-gathered can have many more nonzeros than the column that updates it. We thus usually elect not to store columns in compressed format.

If columns are stored in a spa, then the cost of all the scatters is $\Theta(\text{nnz}(A))$ and the cost of all the gathers is $\Theta(\text{nnz}(L+U))$. This overhead is usually small compared to the cost of floating-point operations, and most sparse LU algorithms incur it.

The other costs in the column-column update loop are indexing overhead to address the spa and insertion of new fill elements into the row lists. In addition, the code also checks for each nonzero in the updating column of L whether the column being updated already has a nonzero in that row (using a bitmap), and if not, the row index is inserted into the column structure and into the row lists. Actual insertions into the row lists cost $\Theta(\text{nnz}(L+U))$ total, as explained above.

4 Comparison with SuperLU

We now compare our out-of-core algorithm to a state-of-the in-core algorithm. SuperLU [7] is block-column left-looking algorithm designed to achieve high performance on machines with data caches. But although the general design objectives of the two algorithms are closely related (to achieve good data reuse in cache or in main memory), the differences

[2]We denote by $\text{nnz}(A)$ the number of nonzeros in A and by $\text{nnz}(U+L)$ the number of nonzeros in the factors of A, assuming that no exact cancellation occurs.

in the performance characteristics of the slow memories (main memory vs. disks) dictate a different design.

The symbolic analysis phases in the two algorithms are similar algorithmically, but ours in designed to perform as little I/O as possible. To achieve this goal we fused loops that read the matrix and implemented an out-of-core sorting algorithm.

Most of the differences, however, are in the numerical factorization phase. Perhaps the most significant difference is the use of a priority queue in our algorithm versus the use of depth-first search (dfs) in SuperLU. To determine which columns of L must update the active panel, SuperLU uses a dfs technique [6]. The graph being searched represents a subset of the nonzeros of L [3]. Earlier algorithms searched in a graph that represents the entire nonzero structure of L, in which case the cost of the searches is proportional to the number of floating-point operations in the algorithm. Searching in the so-called pruned graph [3], as done in SuperLU, reduces the cost of the searches to a small fraction of the cost of floating-point operations. We decided not to use the dfs technique because the pruned graph may be larger than main memory, and searching in a graph stored on disk may be slow. To the best of our knowledge, efficient depth-first searching in a graph that is stored on disk and which grows during the course of the algorithm is not a well-understood problem. Instead, our algorithm uses a priority queue to maintain the set of nonzero rows in the active panel. We cannot provide a simple bound for the cost of the priority queue operations, but their measured cost is small.

Another important difference between SuperLU and our algorithm is that our algorithm maintains a sparse representation of the rows structures in P, whereas SuperLU does not. When SuperLU determines that a column of L must update some columns in the active panel, it cannot determine which columns in the panel must be updated. SuperLU inspects the entire row to determine which column must be updated. Since SuperLU stores the active panel in a dense array, the row is stored in a one-dimensional subarray and it is easy to search it for nonzeros. Also, since in SuperLU the number of columns in a panel is typically small, usually 8–16, so scanning the row is fast. In contrast, in our algorithm the active panel may contain hundreds or thousands of columns, so scanning dense rows can be expensive. Also, when the columns in P are stored in compressed format and not in a dense array, scanning the row cannot be easily done. To summarize, maintaining the row lists allows our algorithm, at a low cost, to avoid unnecessary tests and to store the panel in compressed format.

Also, as explained above, the panel in our algorithm can be larger than memory. In SuperLU the entire panel is stored in memory. Since SuperLU has been shown to achieve high performance with relatively narrow panels, making the panels wider is not important for SuperLU. But for an out-of-core algorithm to achieve high performance, higher levels of data reuse are necessary, and our design enables us to use wider panels without additional I/O.

A significant difference between SuperLU and the current implementation of our algorithm is the fact that SuperLU uses supernode-panel updates whereas our algorithm only uses column-panel updates, which are actually implemented as a sequence of column-column updates. (Supernodes are groups of consecutive columns with similar nonzero structures.) Performing most of the floating-point operations in SuperLU within a supernode-panel update subroutine is an important reason for the high computation rate of SuperLU. Since our algorithm performs the same floating-point operations within a column-column update subroutine at a much slower rate, its overall performance is lower. *This difference, however, is not an essential difference between the two algorithms, and*

supernode-panel updates can be incorporated into our algorithm. As a matter of fact, our algorithm already detects supernodes, but we have not yet implemented a supernode-panel update subroutine.

Most of the essential differences cannot be considered as universal improvements to SuperLU, in the sense that they are unlikely to make SuperLU run faster in core. Instead, these differences represent a design that is more appropriate for the performance characteristics of disks than the design of SuperLU.

5 Implementation

We have implemented the algorithm in C as a Matlab-callable module. Parts of the algorithm that are not performance critical, such as the overall driver routine and the scheduler are currently written in Matlab. We have compiled and used the code under several operating systems, including Sun's Solaris, SGI's Irix, and Microsoft's Windows NT.

We have chosen to implement the code as a collection of C routines callable from Matlab for two reasons. First, this development environment enabled us to quickly prototype key subroutines in Matlab, then convert them to C. The existence of the prototypes in Matlab simplified the debugging phase of the C code, since we could compare the results of runs with the Matlab code to results of runs with the C code. Second, our Matlab-callable solver is easy to use, almost as easy as Matlab's internal solver (the main exception being that the user of our solver must specify a directory for the matrix files). A more flexible way to achieve the ease-of-use goal is to write C or Fortran routines that are independent of Matlab and a Matlab interface, as the implementors of SuperLU and colamd did. This design allows Matlab users to use the software, it allows other users to use the software in independent C and Fortran applications, but it does not provide the implementors with the convenience of developing in the Matlab environment.

We have encountered no problems with porting the code from one Unix platform to another, but we did encounter a problem when we ported the code to Windows NT. Under Windows NT our code cannot open a file in one Matlab-callable C subroutine and use it in another. Each Matlab-callable C subroutine is compiled into a dynamically-linked library called a mex file. Our code normally opens files in one mex file and stores the file descriptors in Matlab data structures. Mex files that are subsequently called retrieve the file descriptors and use them to access the files. We have found that in Windows NT file descriptors are private to each dynamically-linked library, so we had to store file names instead of file descriptors in the Matlab structures and open and close the files in each mex file.

Our algorithm uses matrices stored on disks in compressed column format in files with a special structure. A matrix is stored in one meta-data file and one or more data files. The meta-data file stores the size and location on disk of each column. The location information consists of the index of the data file that contains the column and the offsets within the file of both the vector of row indices and the vector of values. The code automatically splits large matrices into several data files so that no file is larger than about 1Gbytes. We limit files to 1Gbytes so that offsets can be kept in 32-bit integers and so that large matrices can be stored on several file systems. This scheme allows our code to handle matrices that are larger than 4Gbytes on virtually any computer system with sufficient disk space. Although some computer systems directly support files larger than 4Gbytes, 64-bit file offsets, and single files that are stored on several disks, we decided not to rely on such features because many systems do not offer them.

Table 1

The performance of our out-of-core solver on an Origin 200 machine, using one processor and 128Mbytes of main memory. The table shows the time Tsym to perform the symbolic analysis, the time Tnum for the numerical factorization, the time Tcc spent on column-column updates within the numerical factorization phase, the time Tio spent on I/O within the numerical factorization, the number W of columns in P, the computational rate and the size of the factors. Times are reported in seconds.

Matrix	Tsym	Tnum	Tcc	Tio	W	Mflops	Mbytes
twotone	18	1889	1575	187	90	5.1	264
wang4	12	10483	10055	206	143	7.3	470
raefsky4	13	2482	2226	90	188	13.0	320
venkat50	15	674	517	89	59	9.2	229

6 Experimental Results

Experiments that we have performed with the new code indicate that it can easily factor matrices whose factors are larger than memory at reasonable rates. **Matlab was not able to factor any of the test matrices matrices in core on our computers.**

We ran experiments on two machines. The matrices were large nonsymmetric matrices from Davis's matrix collection ([2]; the matrices and their descriptions are available online). The columns of the matrices were ordered using Matlab's colmmd ordering subroutine. Our algorithm was configured in all cases to store the active part P of the panel in a sparse accumulator rather than a compressed column format.

One machine that we used for testing is a 143Mhz Sun UltraSparc workstation with 324MBytes of memory, using a 4GBytes SCSI disk to store matrices and their factors. The solver was allowed to use only 128Mybtes of memory. We used Matlab 5.0. On this machine, Matlab factors sparse matrices that do fit within memory at rates of 6 to 6.5 Mflops. We report on the factorization of two matrices on this machine, vavasis3 and venkat01. Our overall Mflop rates are between 5.27 and 5.47 Mflops. Factoring vavasis3 took about 16,850 seconds, of which less than 700 were spent on I/O. The size of the factors was more than 580 Mbytes. Factoring venkat01 took about 1,040 seconds, of which less than 110 were spent on I/O. The size of the factors was about 220 Mbytes.

We performed additional tests on a 180Mhz SGI Origin 200 computer with two processors, 256Mbytes of main memory, and a 2Gbytes SCSI disk. The solver was allowed to use only 128Mbytes of main memory and used only one out of two processors. We used Matlab 5.2. On this machine, Matlab factors large sparse matrices that do fit within memory (no paging) at rates of 13 to 16 Mflops. It factors dense matrices at rates of 43 Mflops. Our results on this machine are summarized in Table 1. The table shows that only a small fraction of the time is spent on the symbolic analysis phase and on I/O during the numerical factorization phase. Most of the time is spent in column-column updates. The average number of columns of P that a column of L updates when it is loaded from memory ranged from 18 (venkat50) to 66 (raefsky4).

7 Conclusions

We have described an out-of-core sparse LU-factorization algorithm, its implementation, and its performance.

The algorithm is novel in several respects. The algorithm uses a novel schedule that allows panels in a left-looking factorization to be larger than memory without performing additional I/O. The algorithm uses a priority queue to find columns of L that must update the current panel and row lists to find columns in the panel that must be updated.

The main contribution of the algorithm and its implementation is that is allows users to solve very large sparse systems that cannot be solved by existing in-core direct solvers because they run out of memory. Although it is sometimes possible to use virtual memory to factor large matrices with an in-core solver, the performance is usually poor because the paging rate is high. Also, on many computers virtual memory is limited to 2 or 4Gbytes, thereby limiting the size of systems that can be solved. In contrast, our code was designed to perform as little I/O as possible, and it can handle matrices larger than 4Gbytes on virtually any computer.

We plan to implement and describe a supernodal version of this algorithm, which should further improve the performance of the code.

References

[1] P. R. Amestoy, T. A. Davis, and I. S. Duff, *An approximate minimum degree ordering algorithm*, SIAM J. Matrix Anal. Appl., 17(4) (Oct. 1996), pp. 886-905.

[2] T. Davis, *University of Florida sparse matrix collection*, NA Digest, v.92, n.42, Oct. 16, 1994 and NA Digest, v.96, n.28, Jul. 23, 1996, and NA Digest, v.97, n.23, Jun. 7, 1997. available at: http://www.cise.ufl.edu/∼davis/sparse/.

[3] S. C. Eisenstat and J. W. H. Liu, *Exploiting structural symmetry in a sparse partial pivoting code*, SIAM J. Scientific and Statistical Computing, 14 (1993), pp. 253–257.

[4] John R. Gilbert, *Predicting structure in sparse matrix computations*, SIAM J. Matrix Analysis and Applications, 15 (1994), pp. 62–79.

[5] J. R. Gilbert and E. Ng, *Predicting structure in nonsymmetric spase matrix factorizations*, in Alan George, John R. Gilbert, and Joseph W. H. Liu, editors, *Graph Theory and Sparse Matrix Computations*, Springer, 1993.

[6] J. R. Gilbert and T. Peierls, *Sparse partial pivoting in time proportional to arithmetic operations*, SIAM J. Scientific and Statistical Computing, 9 (1988), pp. 862–874.

[7] Xiaoye S. Li, J. Demmel, S. Eisenstat, J. Gilbert, and J. Liu, *A supernodal approach to sparse partial pivoting*, SIAM Journal on Matrix Analysis and Applications, to appear. Previously published as UC Berkeley Tech Report CSD-95-883, September 1995, revised December 1997.