A Supernodal Out-of-Core Sparse Gaussian-Elimination Method*

Sivan Toledo and Anatoli Uchitel

Tel-Aviv University

Abstract. We present an out-of-core sparse direct solver for unsymmetric linear systems. The solver factors the coefficient matrix A into $A = PLU$ using Gaussian elimination with partial pivoting. It assumes that A fits within main memory, but it stores the L and U factors on disk (that is, in files). Experimental results indicate that on small to moderately-large matrices (whose factors fit or almost fit in main memory), our code achieves high performance, comparable to that of SuperLU. In some of these cases it is somewhat slower than SuperLU due to overheads associated with the out-of-core behaviour of the algorithm (in particular the fact that it always writes the factors to files), but not by a large factor. But in other cases it is faster than SuperLU, probably due to more efficient use of the cache. However, it is able to factor matrices whose factors are much larger than main memory, although at lower computational rates.

1 Introduction

We present an out-of-core sparse direct solver for unsymmetric linear systems. The solver factors the coefficient matrix A into $A = PLU$ using Gaussian elimination with partial pivoting. It assumes that A fits within main memory, but it stores the L and U factors on disk (that is, in files).

The availability of computers with large main memories reduced the need for out-of-core solvers. Personal computers can be fitted today with 2-4 GB of main memory for a reasonable cost. This allows users to factor large sparse matrices using high-performance in-core solvers on widely-available machines. But the need to factor matrices that are too large to factor in core still arises occasionally, and new out-of-core algorithms are still being actively developed [11].

The solver that we present here combines ideas from two earlier sparse LU factorization algorithms. The overall structure of our algorithm follows that of SuperLU a high-performance in-core code by Li et al. [4]. We also use techniques from an earlier (and much slower) out-of-core algorithm by Gilbert and Toledo [10].

Experimental results indicate that on small to moderately-large matrices (whose factors fit or almost fit in main memory), our code achieves high performance, comparable to that of SuperLU. In some of these cases it is somewhat slower than SuperLU due to overheads associated with the out-of-core behaviour of the algorithm (in particular the fact that it always writes the factors to files), but not by a large factor. But in other cases it is faster than SuperLU, probably due to more efficient use of the cache. Our method is able, of course, to factor matrices whose factors are much larger than main memory, although at lower computational rates.

* This research was supported by an IBM Faculty Partnership Award, by grant 848/04 from the Israel Science Foundation (founded by the Israel Academy of Sciences and Humanities), and by grant 2002261 from the United-States-Israel Binational Science Foundation.

The paper is organized as follows. Section 2 describes the new algorithm. We discuss the efficiency of the algorithm and its relationship to earlier algorithms in Section 3. Section 4 presents our experimental results and Section 5 our conclusions.

2 The Algorithm and Its Data Structures

This section describes our the algorithm and the data structures that our new factorization code uses.

The algorithm partitions the matrix into a set of consecutive columns that we call *panels*. The algorithm first factors the columns of panel 1, then the columns of panel 2, and so on. The algorithm is left looking (between panels); when it starts processing panel p, the columns of the panels have not been updated at all. During the factorization of the panel, these columns are updated by the columns of panels 1 to $p-1$, and then the panel itself is factored. Once the columns of the panel are factored, they are written to files to make space in memory for the factorization of the next panel.

As we factor a panel, we partition it into *supernodes*, smaller sets of consecutive columns with similar nonzero structure in L. As in SuperLU [4], our algorithm builds supernodes incrementally: after a column is factored, its nonzero structure is inspected to determine whether it should be added to the current supernode. If its nonzero structure is similar to that of the supernode, it is added to it and the algorithm continues to the next column. If it is not similar to the structure of the supernode, the current supernode is finalized and the column starts a new supernode.

External-Memory Data Structures

Our code assumes that the input matrix is stored in main memory in a compressd-column format, although it can be easily converted to use an out-of-core input matrix.

The code stores L and U, as well as the pivoting permutation, in a file or files. It uses the matrix-oriented input-output library developed by Rotkin and Toledo [12]. This library allows the code to write out to files arbitrary rectangular arrays of floating point or integer data and to retrieve such matrices from files. A set of arrays is associated with a symbolic name such as `mymatrix.L`. These arrays can be stored in one file, or if it grows beyond one gigabyte, multiple files. The splitting of the logical data into multiple files is transparent to the code.

We store U column by column in such a file. The columns are compressed, of course. We store L in supernodes, also compressed. The nonzeros of the supernode (padded with zeros if the columns of the supernode do not have identical nonzero structure) are stored in rowwise order, and the rows are ordered in their original order (not in pivot order).

Data Structures

The algorithm uses several data structures. The main data structure stores the columns of the current panel and information about them. Another data structure, a priority queue q, is associated with the entire panel. The algorithm also

uses integer vectors that map columns to their pivot rows (π) and back, and columns to supernodes and back.

The panel data structure consists of an n-by-w array of floating-point numbers, where w is the number of colums in a panel. This is a dense array that stores a sparse submatrix, so it does not use memory as efficiently as possible, but the dense array allows for fast access. The panel is stored in this array in columnwise order (columns are contiguous). The row ordering of panel p reflects all the row exchanges that were made during the factorization of panels 1 through $p-1$. That is, the first row in the array is the pivot row of column 1 and so on.

We associate with each column j in the panel three integer vectors. The vector r_j stores the indices of nonzero rows in column j. The indices are compressed toward the beginning of the vector and an integer stores the number of nonzero rows. The row indices are not sorted. We allocate r_j to be large enough to store all the nonzero indices in column j, using the precomputed bounds on the column counts in L and U. The bound is usually much smaller than n. An n-vector ℓ_j stores the location of indices in r_j. If column j has a zero in a particular row, then the corresponding position of ℓ_j is -1. That is, if row i in the column is nonzero, then element i of ℓ_j stores the position in r_j that contains i, and it contains -1 otherwise. A third vector, h_j, contains a binary heap (a priority queue) of the nonzero row indices in column j. The priority order is the pivoting order. That is, if rows i_1 and i_2 are in the heap, and if i_1 was used as a pivot row before i_2, then i_1 will precede i_2 in the priority ordering. Rows that are nonzero in column j but were not yet used as pivots are not in the queue.

The panel-wide priority queue q stores at most one nonzero row index from every column in the panel. If column j contains nonzeros in rows that were already used as pivots, then q contains a pair (i, j), where i is the minimal row index in j *in the pivot order*. (We actually do not store (i, j) but (s, j), where s is the supernode in which i was used as pivot; this simplifies the code a bit, but complicates the notation here, so we view the contents of the queue as (i, j); i can easily be mapped to s.) If a column contains now nonzero rows that were used as pivots, it is not represented in q.

The Algorithm

The factorization proceeds in phases that each factor one panel. When phase p starts, panels 1 through $p-1$ have already been factored, and the supernodes that they have generated in L are stored in a file. The n-by-w array for the panel contains only -1's, and the ℓ vectors also contain only -1's.

Phase p starts by loading the nonzeors of the columns of the panel from the input-matrix data structure into the panel. We traverse the compressed representation of each column of the panel. We store each nonzero A_{ij} in the n-by-w array, add its row index to r_j, note the location in r_j in the ith position of ℓ_j, and if i has been used as a pivot row, we append it to h_j (ignoring the heap structure that h_j should have). Once all the nonzeros of column j have been processed, we build the heap structure of h_j (see [2]; the cost of building the heap this way is linear in its size, faster than multiple heap insertions).

To finish the pre-processing of column j, we test whether h_j contains any indices. If it does, we append $(\min h_j, j)$ into the panel-wide priority queue, which we will build as a heap later.

The total cost of this panel pre-processing step is linear in the number of nonzeros in the panel in the input matrix A.

Next, the algorithm updates the panel, performing all the column-column operations involving a column from panels 1 to $p-1$ and a column in panel p (the operations are performed in a blocked fashion, as we explain below). This is done as follows. We repeatedly extract a pair (i, j) from q. This tells us that column $\pi^{-1}(i)$ (the column in which row i was the pivot row) must be scaled and subtracted from column j. But in order to perform block operations, we delay the subtraction. Immediately after extracting (i, j) from q, we extract the minimal row index i' in h_j and insert (i', j) to q, to maintain its invariant property. We remember the pair (i, j) in an auxiliary vector and continue to extract pairs from q until the row index in the minimal pair in q belongs to a different supernode than row i, the first row that we extracted from q. Then we stop.

We now read the supernode s of L that contains $\pi^{-1}(i)$ from external memory. We copy the columns of panel p whose indices we extracted from q into a compressed array B. We extract from q row-column pairs; the columns that we copy are the column indices in these pairs. We only copy the rows that appear as row indices in supernode s. In other words, B contains the elements of p that belong to columns that s updates and to rows that appear in s. These numbers are compressed in B, in rowwise order, and the rows have the same ordering as in the compressed supernode s. We can now perform the triangular solve and the matrix-matrix multiplication that constitute the column-column operations involving s and p. Hence, all of these numerical operations are performed by two calls to the level-3 BLAS [8,7]. The supernode s is not needed any more during the processing of panel p, so it can be evicted from memory. The contents of B are now copied back into the main data structure of the panel, updating the r, ℓ, and h vectors where necessary.

When all the column-column operations involving panels 1 through $p-1$ have been applied to p, we factor p itself. At this point the heaps h_j are empty.

The factorization of the panel is supernodal and uses a mixture of left-looking and right-looking updating rules. We first explain the updating rules. Suppose that the next column to be factored is column j and that the factorization of the previous columns in the panel generated supernodes $s_x, \ldots, s_y, s_z$. If the nonzero structure of column j of L will be similar enough to the nonzero structure of s_z, j will join s_z. Therefore, we view s_z as a *pending* supernode, which we store in somewhat different data structure than the *finalized* supernodes $s_x, \ldots, s_y$.

A finalized supernode in the panel is stored in the same data structure as a supernode on disk: compressed in rowwise order (rows are contiguous). A pending supernode is stored in a large two dimensional array. Its rows are contigous, but there are gaps between them. That is, a pending supernode is stored in two dimensional array that has both more rows and more columns than the supernode. This array is partitioned into a U area and an L area. This data structure allows us to add more columns and more nonzero rows to the supernode and to use it as an argument to BLAS subroutines.

When the algorithm reaches the factorization of column j, updates from all the finalized supernodes have already been applied to it (right-looking updates from supernodes within the panels and left-looking updates from earlier panels). We explain below how this happens. But there may be column-column operations

involving the pending supernode that still need to be performed on column j. To find them, we inspect h_j. If it is empty, we do not need to update column j, so we proceed to factor it. If h_j is not empty, we empty it (the indices in it must belong to the pending supernode s_z) and copy the rows in column j which are nonzero in s_z to the compressed buffer B. We then use two level-2 BLAS subroutines [6,5] to update column j and we copy it back to the panel's main data structure, updating r_j and ℓ_j if the column gained new nonzeros.

To factor column j, we use r_j to locate its nonzeros, to find the maximal one (in absolute value), and to scale them. We denote the row index of this maximal element by $\pi(j)$. We then exchange rows j and $\pi(j)$ in all the panel's data structure. We traverse rows j and $\pi(j)$ in the two-dimensional array that stores the panel and swap elements (traversing only columns that have not yet been factored). If both elements (j, k) and $(\pi(j), k)$ are nonzero in some column k, then the only other operation that we need to perform on column k is to insert j into h_k. If only element (j, k) is nonzero, we swap it with the zero in $(\pi(j), k)$ and change the index j somewhere in r_k to $\pi(j)$; this is what we need ℓ_k for, to locate this index with a constant number of iterations. If only element $(\pi(j), k)$ is nonzero, we swap it with the zero in (j, k), change it to j in r_k, and insert j into h_k. In the special case of $j = \pi(j)$, no rows are exchanged, of course, but we still need to traverse row j and to insert a nonzero in (j, k) to h_k. For every column k in the panel, we keep a boolean variable that indicates whether the column is represented in the panel-wide queue q. Whenever we insert an index into h_k we inspect this indicator, and if false, we insert the pair $(\min h_k, k)$ into q. Since we add indices to h_k in pivot order, if already k is represented in q then $(\min h_k, k)$ is already in q. (The scheme can be simplified a bit, but this description keeps the intra-panel updating rules similar to the inter-panel ones, so it is simpler.)

The traversal of the pivot row in the full two-dimensional array storing the panel is somewhat inefficient; we discuss this issue later.

Now that j has been factored, we compare its nonzero structure to that of the pending supernode. If the structures are similar enough (we describe the criteria below), we add j to s_z. This may require adding rows to the nonzero structure of the pending supernode and possibly moving row $\pi(j)$ (if it was already nonzero in s_z) from the L area of the supernode's array to the U area. If the structure of s_z and j are dissimilar or if j is the last column in the panel, we finalize s_z.

To finalize s_z, we first copy it into a compressed two-dimensional array. We then extract from the panel-wide q the indices of all the columns in the panel that s_z must update. We copy these columns to B and use two level-3 BLAS subroutine calls to update them. This operation is identical to updates from supernodes from earlier panels. The finalized supernode can now be written to disk and erased from memory.

The criteria that we use to decide whether a column should be added to a pending supernode are as follows. If the supernode has fewer than 8 columns, we always add the column (to avoid very narrow supernodes). Similarly, we limit the size of supernodes to 80 columns. If the supernode has 8 to 79 columns, we only add the column if the total number of nonzero rows in the supernode is less than 8192.

This concludes the description of the algorithm. We omit discussion of how the panel's data structures are cleared in preparation for the processing of the

next panel; this is done in a conventional way in time proportional to the number of nonzero elements in the panel.

3 Discussion

The following points summarize the main points of the algorithm.

- The columns are statically partitioned into panels, and each panel is dynamically partitioned into supernodes (groups of consecutive columns with similar nonzero structure in L).
- During the factorization of a panel, each factored supernode that updates the panel participates in one supernode-panel update. This operation is performed by two calls to the level-3 BLAS. This rule applies both to supernodes from earlier panels and to supernodes from the same panel.
- Before a supernode is finalized (before the set of columns that belong to it has been determined), the supernode updates one column at a time using supernode-column operations involving two calls to level-2 BLAS subroutines.
- The algorithm uses priority queues to determine which updating operations must be performed.
- The algorithm uses two n-by-w arrays to store the elements of the current panel, as well as a few smaller data structures; one array stores the nonzero values and the other stores the pointer vectors ℓ_j.

There are two places where our algorithm gives up sparsity to obtain higher performance. The first is the use of the n-by-w arrays to store the current panel. This involves a one-time $\Theta(nw)$ cost to set up these arrays initially (to clear them) and it means that the algorithm consumes large amounts of memory if w is large. The second is in the traversal of the full rows j and $\pi(j)$ of the remaining columns of the panel after the factorization of column j. If the panel is very wide (large w), these rows can be fairly sparse. This may cause our algorithm to perform a significant amount of work on zeros. Asymptotically, the total cost of these row scans is $\Theta(\frac{n}{w}w^2) = \Theta(nw)$. Therefore, as long as we use a full array to hold the panel, the row scans are not a dominant cost in the algorithm, due to the cost of initializing the full array.

We did not explore the possibility of compressing the entire panel. Our goal in this research has been to acheive high performance through the use of the level-3 BLAS, and we did not find a way to do this with a compressed panel without a large overhead.

Our algorithm builds on two earlier ones, SuperLU [4], which is an in-core algorithm, and the Gilbert-Toledo sparse out-of-core LU algorithm [10] (we will refer to it as GT). SuperLU uses fixed-width panels and dynamically constructured supernodes. We use variable width panels and dynamically-constructed supernodes. This allows SuperLU to use supernode-panel updates, which are also used in our algorithm. The panels in SuperLU are stored in an n-by-w full data structure; we use the same strategy. However, instead of copying the rows of the panel to a compressed array with the same row structure as the updating supernode, SuperLU performs the updating in-place. The in-place updating of SuperLU saves some overhead and can acheive high data reuse in the registers and caches, but it does not permit the use of standard BLAS libraries that can

be highly optimized. Another difference between our algorithm and SuperLU is that we use level-3 supernode-panel updates within panels (when the source supernode has been finalized; we use supernode-column updates on pending supernodes). This allows us to maintain efficiency even with very wide panels. The wide panels are essential for reducing the amount of I/O traffic. Another difference between our algorithm and SuperLU is that we use priority queues to find supernodes that must update columns in the panel, whereas SuperLU uses the depth-first-search strategy of [9].

As in [10], we prefer the priority-queue approach over the depth-first-search (DFS) approach becasue the DFS repeatedly searches a graph whose size may approach the size of the lower-triangular factor L. The graph may be smaller than L, but there are no useful a priori bounds on its size. Therefore, we conservatively assume that it may not fit in main memory. To avoid searches in a graph stored on disks, we use the priority-queue approach.

The GT algorithm, which is also out-of-core does use priority queues. We also used the overall strategy of GT of using a panel-oriented left-looking approach in an out-of-core LU with partial pivoting. But GT does not use supernodes, so its performance is much worse than that of our algorithm (we demonstrate this below). Our new algorithm does use the out-of-core symbolic analysis phase of GT, mostly as-is.

4 Experimental Results

This section presents experimental results that explore the performance of our new algorithm.

All the experiments that we report on were performed on a 3.2 GHz Intel Pentium 4 computer with 2 GB of main memory running Linux version 2.6.17.

We used GCC version 4.04 to compile all the codes. We linked all the same high-performance implementation of the BLAS, ATLAS version 3.6. We used COLAMD to order the columns of the matrices prior to factoring them. All the codes used strict partial pivoting.

Our out-of-core code stored the factors on a Maxtor 160 GB serial-ATA disk. The factors were stored in files on an EXT2 file system that occupies an 80 GB partition on the disk. The same disk also hosts a 4 GB swap partition, but nothing else (the rest of the disk was not used during these runs). The machine was dedicated to these experiments and did not run any significant process during the experiments.

We used for the experiments the 22 matrices that are described in Table 1. The matrices are all available from Tim Davis's sparse matrix collection[1]. The table shows the size of the matrices, the size of the computed factors, the size of panels that our code chose (automatically), and the factorization times.

The table shows that the code was able to factor a matrix whose factors are much larger than main memory, sparsine. Its L and U factors have roughly the same number of nonzeros, more than 500 million in each. Just the values of the nonzeros in L take more than 4 GB to store, twice the amount of main memory on the machine that factored the matrix. The code ran for more than 52 hours.

[1] `http://www.cise.ufl.edu/research/sparse/matrices/`

Table 1. The matrices that we used for the experiments.

Name	Order	1000's of nonzeros in A	1000's of nonzeros in L and U	Max columns in panel	OOC factor time (sec)
1 psmigr_1	3140	543	9181	15846	15.1
2 raefsky4	19779	1329	20142	2516	15.2
3 raefsky3	21200	1489	22628	2347	16.3
4 rim	22560	1015	20340	2206	17.1
5 ex11	16614	1097	21067	2995	17.2
6 fidap011	16614	1091	21938	2995	20.7
7 zhao2	33861	166	20627	1470	26.4
8 twotone	120750	1224	27543	413	30.8
9 fidapm11	22294	626	30134	2232	31.2
10 wang4	26068	177	29107	1909	41.2
11 cage10	11397	151	27399	4366	52.8
12 bbmat	38744	1772	53961	1285	63.9
13 av41092	41092	1684	47863	1211	83.2
14 mark3jac140	64089	400	53891	777	95.6
15 xenon1	48600	1181	66641	1024	107.2
16 g7jac200	59310	838	65948	839	119.4
17 li	22695	1350	71274	2193	167.8
18 ecl32	51993	380	82588	957	171.0
19 gupta3	16783	9323	82202	2965	227.6
20 xenon2	157464	3867	387103	316	1644.9
21 cage11	39082	560	285699	1274	3620.4
22 sparsine	50000	1549	1105720	996	187296.7

We were not able to find in sparse matrix collections additional matrices whose factors are significantly larger than main memory.

The factorization of the largest matrix, sparsine, ran at much lower computation rates than the factorization of the smaller matrices. The factorization of this matrix required a large amount of IO that slowed down the code by about a factor of five.

Our code chose the panel size as follows. It broke the column elimination tree into subtrees. The size of each subtree was limited by two constraints: a maximal number of columns and a maximal number of predicted nonzeros in the corresponding columns of L and U. The limit on the number of columns was chosen so that the total size of the n-by-w array is at most 3/16 of the available memory. The total number of predicted nonzeros in a subtree was limited to $820,000$. This is a fairly small number compared to the total amount of memory. We selected this number experimentally to achive high performance. This is the only tunable parameter in the code.

Figure 1 compares the running times of our new out-of-core code to those of three other sparse LU factorization codes that all perform partial pivoting. One code is the out-of-core code of Gilbert and Toledo [10]. Another is an in-core unsymmetric-pattern multifrontal code [1] which is now part of TAUCS, a library of sparse linear solvers that our group has developed. This code, denoted MULTILU below, is based on the UMFPACK 4.0 algorithm [3]. The last code that we use in this comparison is SuperLU [4], also an in-core algorithm. Our new code is denoted in the figures by TSOOC. In these experiments, we used fixed-size panels (fixed number of columns) in our code.

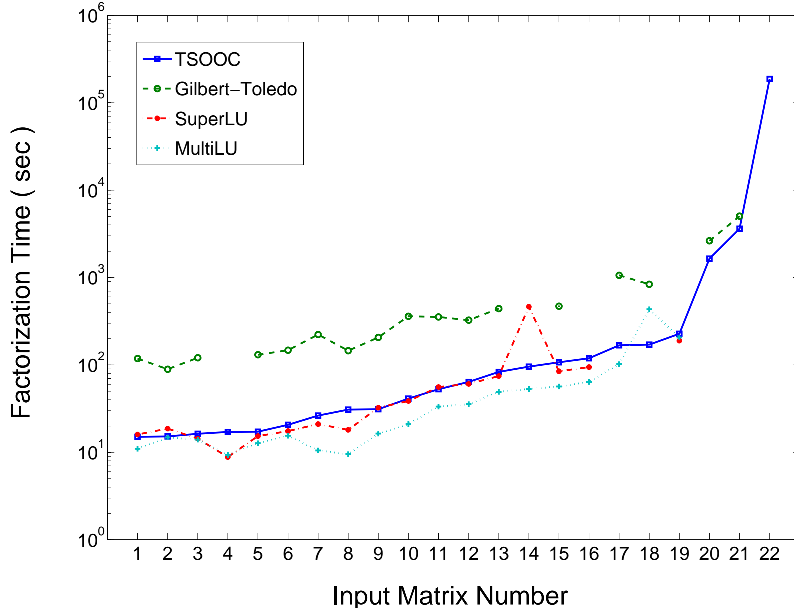

Fig. 1. The factorization times of our new code (TSOOC), along with the factorization times of several existing codes: the out-of-core code of Gilbert and Toledo, the in-core unsymmetric multifrontal code in TAUCS, denoted MultiLU in the graphs), and SuperLU. The ordering of the matrices is according to the ordering in Table 1.

The results show that our new code is usually slower than SuperLU and MULTILU when these codes can factor the matrix, but not by a large factor. In many cases the performance of our code is similar to that of SuperLU, but in two cases SuperLU is more than twice as fast as our out-of-core code. In these experiments, MULTILU is usually faster than SuperLU, and on one matrix it is five times faster than our code. But on most matrices our code is within a factor of 2.5 of the performance of both SuperLU and MULTILU. On one matrix SuperLU was much slower than our code and on another MultiLU was much slower, but these behaviors are likely to have been caused by excessive paging. Both codes failed to factor some of the larger matrices due to lack of sufficient memory to factor them in core.

The code of Gilbert and Toledo managed to factor some of the matrices that were too large for SuperLU and MULTILU, but it was slower than the other codes by a large factor (more than a factor of 8 slower than our new out-of-core code), and it also failed on matrices that the other codes managed to factor in core.

Figure 2 shows the influence of the panel width on the performance of our code on one matrix; experiments on other matrices led to graphs with the same structure, sometimes with some fluctuations near the performance peak. Performance first rises sharply when the panel size grows, and it eventually drops slowly as it becomes very large.

5 Conclusions

We have presented a new out-of-core sparse LU factorization algorithm. The performance of the new algorithm degrades gracefully as the problem gets larger. The code factored two matrices that in-core codes failed to factor at computa-

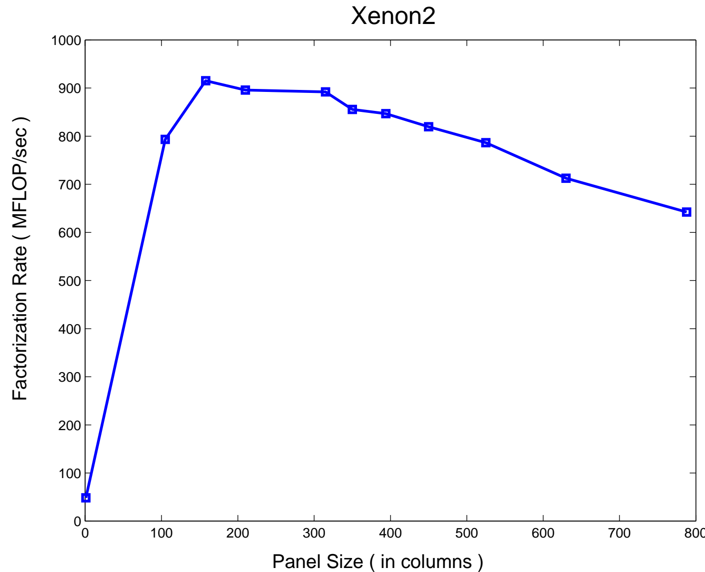

Fig. 2. Performance of our algorithm as a function of the width of the panel on the matrix Xenon2.

tional rates (719 and 510 Mflop/s) that are comparable to the rates it achieved on small matrices . Only on the largest matrix, sparsine, the performance of the algorithm declined considerably, 90 Mflop/s, due to IO.

References

1. Haim Avron, Gil Shklarski, and Sivan Toledo. Parallel unsymmetric-pattern multifrontal sparse LU with column preordering. Submitted for publication in ACM Transactions on Mathematical Software, December 2004.
2. Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT Press and McGraw-Hill, 2nd edition, 2001.
3. Timothy A. Davis. A column pre-ordering strategy for the unsymmetric-pattern multifrontal method. *ACM Trans. Math. Softw.*, 30(2):165–195, 2004.
4. James W. Demmel, Stanley C. Eisenstat, John R. Gilbert, Xiaoye S. Li, and Joseph W. H. Liu. A supernodal approach to sparse partial pivoting. *SIAM Journal on Matrix Analysis and Applications*, 20:720–755, 1999.
5. Jack J. Dongarra, Jeremy Du Cruz, Sven Hammarling, and Ian Duff. Algorithm 656: An extended set of Fortran basic linear algebra subprograms. *ACM Transactions on Mathematical Software*, 14:18–32, 1988.
6. Jack J. Dongarra, Jeremy Du Cruz, Sven Hammarling, and Ian Duff. An extended set of Fortran basic linear algebra subprograms. *ACM Transactions on Mathematical Software*, 14:1–17, 1988.
7. Jack J. Dongarra, Jeremy Du Cruz, Sven Hammarling, and Ian Duff. Algorithm 679: A set of level 3 basic linear algebra subprograms. *ACM Transactions on Mathematical Software*, 16(1):18–28, 1990.
8. Jack J. Dongarra, Jeremy Du Cruz, Sven Hammarling, and Ian Duff. A set of level 3 basic linear algebra subprograms. *ACM Transactions on Mathematical Software*, 16(1):1–17, 1990.
9. John R. Gilbert and Tim Peierls. Sparse partial pivoting in time proportional to arithmetic operations. *SIAM Journal on Scientific and Statistical Computing*, 9:862–874, 1988.
10. John R. Gilbert and Sivan Toledo. High-performance out-of-core sparse LU factorization. In *Proceedings of the 9th SIAM Conference on Parallel Processing for Scientific Computing*, San-Antonio, Texas, 1999. 10 pages on CDROM.
11. John K. Reid and Jennifer A. Scott. The design of an out-of-core multifrontal solver for the 21st century. In *Proceedings of the Workshop on State-of-the-Art in Scientific and Paralell Computing*, Umea, Sweden, June 2006.
12. Vladimir Rotkin and Sivan Toledo. The design and implementation of a new out-of-core sparse Cholesky factorization method. *ACM Transactions on Mathematical Software*, 30:19–46, 2004.