Efficient Sparse LU Factorization with Lazy Space Allocation

Bin Jiang[*], Steven Richman[†], Kai Shen[‡], Tao Yang[§]

Abstract

Static symbolic factorization coupled with 2D supernode partitioning and asynchronous computation scheduling is a viable approach for sparse LU with dynamic partial pivoting. Our previous implementation, called S^+, uses those techniques and achieves high gigaflop rates on distributed memory machines. This paper studies the space requirement of this approach and proposes an optimization strategy called lazy space allocation which acquires memory on-the-fly only when it is necessary. This strategy can effectively control memory usage, especially when static symbolic factorization overestimates fill-ins excessively. Our experiments show that the improved S^+ code, which combines this strategy with elimination-forest guided partitioning and scheduling, has sequential time and space cost competitive with SuperLU, is space scalable for solving problems of large sizes on multiple processors, and can deliver up to 10 GFLOPS on 128 Cray 450Mhz T3E nodes.

1 Introduction

Solution of sparse linear systems is a computational bottleneck in many problems. When dynamic pivoting is required to maintain numerical stability in using direct methods for solving non-symmetric linear systems, it is hard to develop high performance parallel code because partial pivoting causes severe caching miss and load imbalance on modern architectures with memory hierarchies. The previous work such as SuperLU [5] has addressed parallelization using shared memory platforms. For distributed memory machines, in [9, 10] we proposed an approach that adopts a static symbolic factorization scheme [12] to avoid data structure variation, identifies data regularity to maximize the use of BLAS-3 operations, and utilizes graph scheduling techniques and efficient run-time support [11] to exploit irregular parallelism.

Recently [16] we have further studied the properties of elimination forests to guide supernode partitioning/amalgamation and execution scheduling. The new code with 2D mapping, called S^+, effectively clusters dense structures without introducing too many zeros in the BLAS computation, and uses supernodal matrix multiplication to retain the BLAS-3 level efficiency and avoid unnecessary arithmetic operations. The experiments show that S^+ improves our previous code substantially and can achieve up to 11.04GFLOPS on 128 Cray 450MHz T3E nodes.

Our previous evaluation shows that for most of tested matrices, static symbolic factorization provides fairly accurate prediction of nonzero patterns and only creates 10% to

[*]Department of Computer Science, University of California Santa Barbara, CA 93106. hjiang@cs.ucsb.edu

[†]Department of Computer Science, University of California Santa Barbara, CA 93106. joy@cs.ucsb.edu

[‡]Department of Computer Science, University of California Santa Barbara, CA 93106. kshen@cs.ucsb.edu

[§]Department of Computer Science, University of California Santa Barbara, CA 93106. tyang@cs.ucsb.edu

50% more fill-ins compared to dynamic symbolic factorization used in SuperLU. However, for some matrices static symbolic factorization creates too many fill-ins and our previous solution does not provide a smooth adaptation in handling such cases. For these cases, we find that the prediction can contain a significant number of fill-ins that remain zero throughout numerical factorization. This indicates that space allocated to those fill-ins is unnecessary.

Thus our first space-saving strategy is to delay the space allocation decision and acquire memory only when a submatrix block becomes truly nonzero during numerical computation. Such a dynamic space allocation strategy can lead to a relatively small space requirement even if static factorization excessively over-predicts nonzero fill-ins. Another strategy we have proposed is to examine if space recycling for some nonzero submatrices is possible since a nonzero submatrix may become zero during numerical factorization due to pivoting. This has the potential to save significantly more space since the early identification of zero blocks prevents their propagation in the update phase of the factorization. In the rest of this paper, we will use S^+ to denote the original S^+ code and use $LazyS^+$ to denote the improved S^+ by using proposed space optimization strategies.

The rest of this paper is organized as follows. Section 2 gives the background knowledge of sparse LU factorization. Section 3 presents two space optimization strategies. Section 4 describes the experimental results of sequential performance. Section 5 presents the experimental results of parallel performance. Section 6 concludes the paper.

2 Background

Static symbolic factorization. Static symbolic factorization is proposed in [12] to identify the worst case nonzero patterns without knowing numerical values of elements. The basic idea is to statically consider all the possible pivoting choices at each elimination step and the space is allocated for all the possible nonzero entries. Using an efficient implementation of the symbolic factorization algorithm [13], this preprocessing step can be very fast. For example, it costs less than one second for most of our tested matrices, at worst it costs 2 seconds on a single node of Cray T3E, and the memory requirement is relatively small. The dynamic factorization, which is used in the sequential and share-memory versions of SuperLU [14], provides more accurate data structure prediction on the fly, but it is challenging to parallelize SuperLU with low runtime control overhead on distributed memory machines.

In [9, 10], we show that static factorization does not produce too many fill-ins for most of the tested matrices, even for large matrices using a simple matrix ordering strategy (minimum degree ordering). For few tested matrices, static factorization generates an excessive amount of fill-ins which in turn costs a large amount of space and time for LU factorization. In this paper we will use lazy allocation to reduce the actual usage of space for those matrices.

Elimination forests. Considering an $n \times n$ sparse matrix A, we assume that every diagonal element of A is nonzero. Notice that for any nonsingular matrix which does not have a zero-free diagonal, it is always possible to permute the rows of the matrix so that the permuted matrix has a zero-free diagonal [7]. In [16], we propose the following definitions which will also be used in the rest of this paper. We still call the matrix after symbolic factorization as A since this paper assumes the symbolic factorization is conducted first. Let $a_{i,j}$ be the element of row i and column j in A and $a_{i:j,s:t}$ be the submatrix of A from row i to row j and column s to t. Let L_k denote column k of the L factor, which is $a_{k:n,k:k}$.

Let U_k denote row k of the U factor, which is $a_{k:k,k:n}$. Also let $|L_k|$ and $|U_k|$ be the total number of nonzeros and fill-ins in those structures.

DEFINITION 2.1. *An LU Elimination forest for an $n \times n$ matrix A has n vertices numbered from 1 to n. For any two vertices k and j ($k < j$), there is an edge from vertex j to vertex k in the forest if and only if a_{kj} is the first off-diagonal nonzero in U_k and $|L_k| > 1$. Vertex j is called the parent of vertex k, and vertex k is called a child of vertex j.*

An elimination forest for a given matrix can be generated in a time complexity of $O(n)$ and it can actually be a byproduct of the symbolic factorization. The following two theorems demonstrate the properties of an LU elimination forest. Theorem 2.1 captures the structural containment between two columns in L and two rows in U, which will be used for efficient supernode partitioning and amalgamation which is described in the next subsection. Theorem 2.2 indicates data dependencies in the numerical elimination, which can guide our parallel code in scheduling asynchronous parallelism. The details of the analysis are in [16].

THEOREM 2.1. *If vertex j is the ancestor of vertex k (i.e., there is a path from vertex j to vertex k) in the elimination forest, then $L_k - \{k, k+1, \cdots, j-1\} \subseteq L_j$ and $U_k - \{k, k+1, \cdots, j-1\} \subseteq U_j$.*

THEOREM 2.2. *L_j will be used to directly or indirectly update L_i in LU factorization if and only if vertex i is an ancestor of vertex j in the elimination forest.*

2D L/U supernode partitioning and amalgamation. After the nonzero fill-in pattern of a matrix is predicted, the matrix is further partitioned using a supernodal approach to improve the caching performance. In [14], a nonsymmetric supernode is define as a group of consecutive columns in which the corresponding L factor has a dense lower triangular block on the diagonal and the same nonzero pattern below the diagonal. Based on this definition, in each column block the L part only contains dense subrows. We call this partitioning method L supernode partitioning. Here by "subrow" we mean the contiguous part of a row within a supernode.

After an L supernode partition has been obtained on a sparse matrix A, the same partitioning is applied to the rows of the matrix to further break each supernode into submatrices. This is also known as U supernode partitioning. In [10], we show that after the L/U supernode partitioning, each diagonal submatrix is dense, and each nonzero off-diagonal submatrix in the L part contains only dense subrows, and furthermore each nonzero submatrix in the U factor of A contains only dense subcolumns. This is the key to maximize the use of BLAS-3 subroutines [6] in our algorithm. And on most current commodity processors with memory hierarchies, BLAS-3 subroutines usually outperform BLAS-2 subroutines substantially when implementing the same functionality [6]. In [16], we further show that supernode partitioning can be performed in time complexity $O(n)$ by using elimination forests.

Figure 1 illustrates an example of a partitioned sparse matrix and the black areas depict dense submatrices, subrows and subcolumns.

Another technique called amalgamation can be applied after supernode partitioning to further increase the supernode size and improve the caching performance. This can be done in time complexity $O(n)$ by using the properties of elimination forests and are also very effective [16].

2D data mapping and asynchronous parallelism exploitation. Given an $n \times n$ matrix A, assume that after the matrix partitioning it has $N \times N$ submatrix blocks. For example, the matrix in Figure 1 has 8×8 submatrices. Let $A_{i,j}$ denote a submatrix of A with

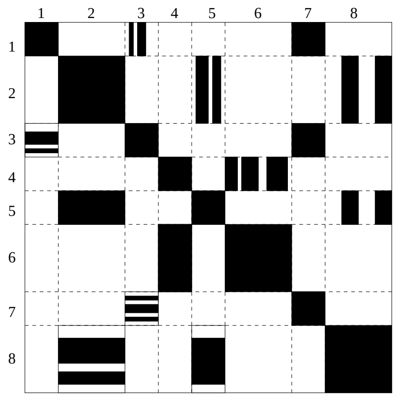

FIG. 1. *Example of a partitioned sparse matrix.*

row block index i and column block index j. We use 2D block-cyclic mapping: processors are viewed as a 2D grid, and a column block of A is assigned to a column of processors. 2D sparse LU Factorization is more scalable than the 1D data mapping [8]. However 2D mapping introduces more overhead for pivoting and row swapping. Each column block k is associated with two types of tasks: $Factor(k)$ and $Update(k, j)$ for $1 \leq k < j \leq N$. 1) Task $Factor(k)$ factorizes all the columns in the k-th column block, including finding the pivoting sequence associated with those columns and updating the lower triangular portion of column block k. The pivoting sequence is held until the factorization of the k-th column block is completed. Then the pivoting sequence is applied to the rest of the matrix. This is called "delayed pivoting" [3]. 2) Task $Update(k, j)$ uses column block k $(A_{k,k}, A_{k+1,k}, \cdots, A_{N,k})$ to modify column block j. That includes "row swapping" using the result of pivoting derived by $Factor(k)$, "scaling" which uses the factorized submatrix $A_{k,k}$ to scale $A_{k,j}$, and "updating" which uses submatrices $A_{i,k}$ and $A_{k,j}$ to modify $A_{i,j}$ for $k+1 \leq i \leq N$. Figure 2 outlines the partitioned LU factorization algorithm with partial pivoting.

```
for k = 1 to N
    Perform task Factor(k);
    for j = k + 1 to N with A_kj ≠ 0
       Perform task Update(k, j);
    endfor
endfor
```

FIG. 2. *Partitioned sparse LU factorization with partial pivoting.*

In [16], we have proposed an asynchronous scheduling guided by elimination forest. This strategy enables the parallelism exploitation among $Factor()$ tasks which used to be serialized by previous scheduling strategies.

3 Space Optimization Techniques

As we mentioned in Section 1, static symbolic factorization may produce excessive amount of fill-ins for some test matrices. This makes our S^+ LU factorization very space and time consuming for these matrices. How to save space and speed up LU for these matrices becomes a very serious problem for us. In this section, we introduce two techniques to solve this problem. The first technique, called delayed space allocation, delays the allocation of space for a block until some of its elements truly becomes nonzero. The second technique, called space reclamation, deallocates space for previously nonzero blocks which become zeros at some step of the factorization.

3.1 Delayed space allocation

Since symbolic factorization can introduce many more fill-ins than the nonzeros of the original matrix, the blocks produced during L/U supernode partitioning basically are of the following three types:

1. Some elements in a block are nonzeros in the original matrix. For this type of blocks, we should allocate the space for it in advance.

2. All the elements in a block are zeros in the original matrix during the initialization, but some elements become nonzeros during the numerical factorization. For this type of blocks, we don't allocate space for them at first and will allocate space when nonzero elements are produced later on.

3. All the elements in the block are zeros in the original matrix during the initialization, and remain zeros throughout numerical factorization. For this type of blocks, we should not allocate space for them.

Our experiments showed that the matrices on which S^+ code didn't run well (i.e., S^+ needed a lot of space and time) contain $10-24\%$ of type 3 blocks, i.e., blocks which always remain zero from beginning to end. In S^+, these blocks occupied space and were involved in the numerical factorization even though they did nothing, thereby wasting a lot of time and space.

Therefore we use different space allocation policies for different types of blocks in the matrices. The general idea is to delay the space allocation decision and acquire memory only when a block becomes truly nonzero during numerical computation. Such a dynamic space allocation strategy can lead to a relatively small space requirement even if static factorization excessively over-predicts nonzero fill-ins. We discuss the impact of this strategy in the following aspects:

- For the relatively dense matrices, this strategy has almost the same effect as without using it since almost all the blocks produced at the step of supernode partitioning contain at least some nonzeros in it or will have some nonzeros during numerical factorization, the number of blocks of type 3 is very small. Thus lazy allocation won't save a lot of space for those matrices.

- However for the relatively sparse matrices which contain many blocks of types 3, the lazy allocation technique will never allocate the space for those blocks of type 3. The space saving is obvious.

- Further savings can be reaped in another part of our code: numerical factorization. First of all, each $Factor$ task in numerical factorization needs to factorize one column block. And all zero blocks are unnecessary to get involved into this task. But as long as a block is recognized as a nonzero block in numeric factorization, S^+ still ran it even though it may be actually a zero block during numeric factorization. However, in $LazyS^+$ with delayed space allocation, those actually zero blocks are not allocated space throughout the numerical factorization and they will be treated as zero blocks without getting involved into the numerical factorization. The *Update* tasks are the most time consuming part of numerical factorization. $Update(k, j)$ uses blocks $A_{i,k}$ and $A_{k,j}$ to update block $A_{i,j}$ for every $k < i \leq N$. If either $A_{i,k}$ or $A_{k,j}$ is a zero block, it is unnecessary to update block $A_{i,j}$ in this task (see Figure 3). However, S^+ code updated every $A_{i,j}$ if both $A_{i,k}$ and $A_{k,j}$ are recognized as nonzero blocks by symbolic factorization even though one of them is a zero block during numeric factorization. Therefore a lot of time was wasted in unnecessary updating. $LazyS^+$ with delayed space allocation gets rid of this shortcoming. It first checks block $A_{k,j}$. If it is a zero block, the whole $Update(k, j)$ task is skipped (see Figure 4(a)). Otherwise, is picks up the nonzero blocks $A_{i,k}$ in column k, and update the corresponding blocks $A_{i,j}$ (see Figure 4(b)).

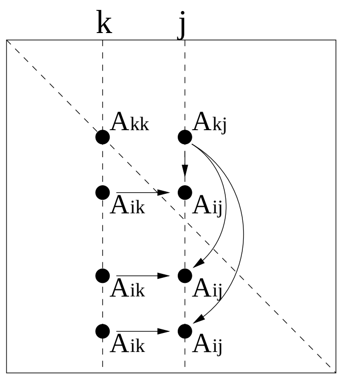

FIG. 3. *Illustration of* $Update(k, j)$ *task.*

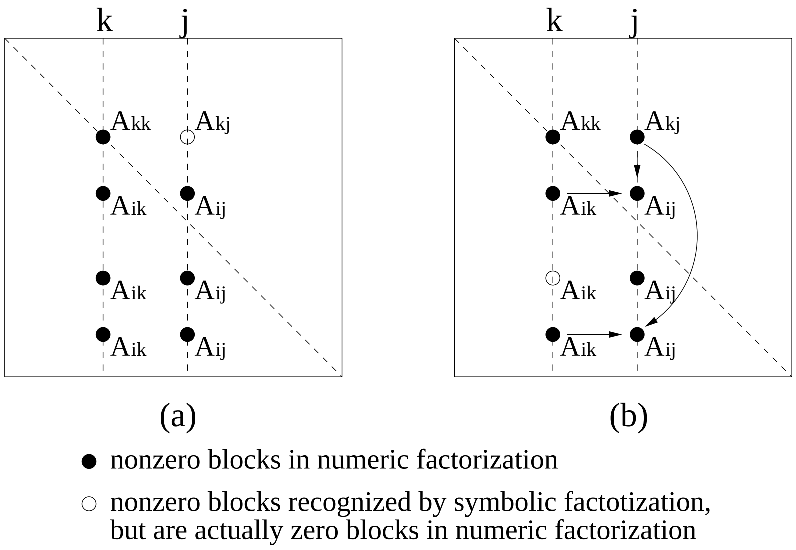

FIG. 4. *Illustration of* $Update(k, j)$ *task with delayed space allocation.*

3.2 Space reclamation

Our experiments also show that some nonzero blocks which have been assigned space will become zero blocks later on due to pivoting. Since zero blocks don't need space any more, we can collect the space of these blocks. Therefore we can save the space they occupied. Furthermore, these blocks won't appear in future $Factor(k)$ and $Update(k, j)$ tasks which saves unnecessary computation time. This is our second strategy of space optimization.

The execution of task $Update(k, j)$ uses blocks $A_{i,k}$ and $A_{k,j}$ to update block $A_{i,j}$ for every $k < i \leq N$. If block $A_{i,k}$ has been allocated space earlier due to some nonzero elements in it but at this time contains only zeros due to pivoting, the benefits of this space reclamation strategy are considerable in several ways. Without this strategy, $A_{i,k}$ would still be treated as a nonzero block, and it would still get involved in task $Update(k, j)$ which is actually unnecessary. Furthermore, if the block $A_{i,j}$ has not been allocated space before, this unnecessary update would enforce a space allocation for $A_{i,j}$ which is again unnecessary. In the worst case, this situation would propagate along with the factorization process and produce a considerable amount of wasted space and unnecessary computation. The space reclamation strategy gets rid of this problem by checking if some formerly-nonzero blocks on column block k or $A_{k,j}$ have become zero in the beginning of task $Update(k, j)$. If they have, their space will be deallocated and those blocks will also be excluded from future computation.

4 Experimental Studies on Sequential Performance

The sequential machine we use is a SUN 167MHZ Ultra-1 with 320MB memory, 16KB L1 data cache and 512KB L2 cache. We have compared our sequential code with SuperLU, but not UMFPACK [2] because SuperLU has been shown competitive to UMFPACK [4]. Table 1 lists the testing matrices which come from various application domains. All matrices are ordered using the minimum degree algorithm on $A^t * A$ matrix for a given matrix A. In computing gigaflop rates, we use operation counts reported by SuperLU for the tested matrices, which excludes the extra computation introduced by static symbolic factorization.

TABLE 1
Testing matrices and their application domains

Matrix	Application Domain
sherman3, sherman5, orsreg1, saylr4	oil reservoir simulation
goodwin	fluid mechanics problem
e40r0100	fluid dynamics
raefsky4	buckling for a container model
fidap011	3-D steady flow calculation
af23560, vavasis	PDE with varying coefficients
TIa, TIb, TId, memplus, wang3	circuit and device simulation

A Comparison of S^+, $LazyS^+$ and SuperLU. The sequential performance of S^+, SuperLU and $LazyS^+$ on the testing matrices is listed in Table 2. This result shows that the space optimization strategies of $LazyS^+$ are effective for our testing matrices, especially for circuit/device simulation matrices TIa, TId, memplus, TIb and wang3. Note that S^+ and SuperLU cause paging on matrices TIb and wang3 due to excessive space requirement. In computing the average saving on time cost, we exclude the paging effects by not counting matrices TIb and wang3.

The results on Ultra-1 show that on average, $LazyS^+$ uses 17.5% less space compared to S^+. Compared to SuperLU, our algorithm actually uses 3.9% less space on average while static symbolic factorization predicts 38% more nonzeros. Notice that the space cost in our evaluation includes symbolic factorization. This part of cost ranges from 1% to 7% of the total cost. In terms of average time cost, $LazyS^+$ is about 27% faster than SuperLU, which is consistent to the results in [16]. The reason is that we use a fast supernodal matrix multiplication kernel which accommodates sparse space structure and has time performance similar to BLAS-3. We also use the newly-developed partitioning techniques based on elimination-forests. It is possible that both $LazyS^+$ and SuperLU can be further tuned to obtain better time and space performance. Thus the above measurements are illustrative in assessing the competitiveness of our approach. Notice that time measurement excludes symbolic preprocessing time; however, symbolic factorization in our algorithms is very fast and takes only about 3.4% of numerical factorization time. In Table 2, the data inside the parenthesis behind time of $LazyS^+$ indicates the time for symbolic factorization in $LazyS^+$ and S^+.

TABLE 2

Sequential performance on SUN Ultra-1. A "-" implies the data is not available due to insufficient memory. Time is in seconds and space is in MBytes.

Matrix	S^+		SuperLU		$LazyS^+$		Exec. Time Ratio	
	Time	Space	Time	Space	Time	Space	$\frac{LazyS^+}{SuperLU}$	$\frac{LazyS^+}{S^+}$
sherman5	0.97	3.061	1.09	3.305	0.93 (0.06)	2.964	0.853	0.959
sherman3	2.33	5.713	2.15	5.412	2.20 (0.11)	5.536	1.023	0.944
orsreg1	2.37	5.077	2.12	4.555	1.95 (0.08)	4.730	0.920	0.823
saylr4	4.02	8.509	3.63	7.386	3.50 (0.14)	8.014	0.964	0.870
goodwin	15.02	29.192	22.71	35.555	14.91 (0.42)	28.995	0.657	0.993
e40r0100	52.87	79.086	77.89	93.214	54.78 (0.90)	78.568	0.703	1.036
raefsky4	658.89	303.617	857.67	272.947	606.72 (2.33)	285.920	0.707	0.921
af23560	159.62	170.166	180.65	147.307	157.04 (1.49)	162.839	0.869	0.984
fidap011	357.26	221.074	683.64	271.423	360.62 (1.97)	219.208	0.528	1.009
TIa	6.31	8.541	5.88	6.265	3.97 (0.25)	7.321	0.675	0.629
TId	30.03	29.647	30.08	18.741	11.00 (0.69)	19.655	0.366	0.366
memplus	344.64	138.218	322.36	75.194	44.21 (1.81)	68.467	0.137	0.128
TIb	1325.29	341.418	1360.58	221.285	73.97 (4.19)	107.711	0.054	0.056
wang3	1431.91	430.817	-	-	645.15 (2.72)	347.505	-	0.451

Sensitiveness on block size limit. The above experiments use the block size limit 25. Table 3 shows the performance of $LazyS^+$ under different block size limit. For most of matrices where fill-in overestimation is not excessive,, when we reduce this limit to 20, 15, 10, and 5, changes in space saving are insignificant while processing time increase gradually due to degradation of caching performance. For matrices with high fill-in overestimation, space saving is more effective when the block size is reduced. The reason is that when the block size become smaller, the probability of a block being zero block is higher. Therefore the lazy allocation strategy in $LazyS^+$ will become more effective.

Effectiveness of space optimization. We also conducted experiments concerning the effectiveness of two space optimization strategies. Table 4 shows the time and space performance of S^+, $LazyS^+$ and $LazyS^+$ without space reclamation. We can see that the space reclamation plays a more important role than delayed allocation in the overall improvement. And on average, the delayed allocation alone saves 0.4% in time and 1.6%

TABLE 3

Sequential performance of LazyS$^+$ with different block size limits. Time is in seconds and space is in MBytes.

Matrix	25		20		15		10		5	
	Time	Space	Time	Space	Time	Space	Time	Space	Time	Space
sherman5	0.93	2.964	0.99	2.963	1.12	2.939	1.37	2.953	3.06	3.119
sherman3	2.20	5.536	2.27	5.500	2.79	5.545	3.52	5.600	7.92	5.988
orsreg1	1.95	4.730	1.94	4.681	2.23	4.558	2.31	4.308	3.34	4.128
saylr4	3.50	8.014	3.55	7.957	4.09	7.834	4.41	7.498	6.80	7.119
goodwin	14.91	28.995	16.61	29.224	20.89	29.168	28.02	29.725	79.10	32.760
e40r0100	54.78	78.568	58.92	78.733	76.45	79.170	102.39	80.373	334.93	88.968
raefsky4	606.72	285.920	627.86	283.803	809.11	279.931	997.88	278.565	2605.30	289.890
af23560	157.04	162.839	156.13	161.196	195.16	158.626	233.53	155.907	594.58	159.475
fidap011	360.62	219.208	388.88	219.622	513.05	220.791	688.81	225.337	2082.15	251.426
TIa	3.97	7.321	3.71	7.078	3.60	6.668	3.38	5.912	3.88	5.248
TId	11.00	19.655	9.48	17.551	8.29	15.090	7.50	12.641	7.77	12.410
memplus	44.21	68.467	32.73	65.670	29.19	62.752	19.35	56.590	17.98	41.666
TIb	73.97	107.711	66.22	94.074	62.29	79.032	70.74	66.089	185.74	84.377
wang3	645.15	347.505	588.30	331.346	627.20	310.836	602.84	282.690	952.76	270.136

in space while the two strategies combined saves 21.0% in time and 17.5% in space. Note that we exclude the paging effect in this calculation, i.e., not counting matrices TIb and wang3 when calculating savings on time.

TABLE 4

Effectiveness of individual lazy strategy. Time is in seconds and space is in MBytes. The percentage beside each number is the improvement over S$^+$.

Matrix	*LazyS*$^+$		*LazyS*$^+$ without space reclamation		S^+	
	Time	Space	Time	Space	Time	Space
sherman5	0.93 (4%)	2.964 (3.2%)	0.99 (-2%)	3.018 (1.4%)	0.97	3.061
sherman3	2.20 (5.6%)	5.536 (3.1%)	2.36 (-1.3%)	5.657 (0.98%)	2.33	5.713
orsreg1	1.95 (17.8%)	4.730 (6.8%)	2.33 (1.7%)	5.038 (0.77%)	2.37	5.077
saylr4	3.50 (12.9%)	8.014 (5.8%)	4.03 (-0.25%)	8.418 (1.1%)	4.02	8.509
goodwin	14.91 (0.73%)	28.995 (5.8%)	15.42 (-0.25%)	29.070 (0.41%)	15.02	29.192
e40r0100	54.78 (-3.6%)	78.568 (0.65%)	55.07 (-4.2%)	78.761 (0.41%)	52.87	79.086
raefsky4	606.72 (7.9%)	285.920 (5.8%)	672.02 (-2%)	300.094 (1.2%)	658.89	303.617
af23560	157.04 (1.6%)	162.839 (4.3%)	163.45 (-2.4%)	167.710 (1.4%)	159.62	170.166
fidap011	360.62 (-0.9%)	219.208 (0.8%)	364.65 (-2%)	219.422 (0.7%)	357.26	221.074
TIa	3.97 (37%)	7.321 (14%)	5.67 (10%)	8.133 (4.8%)	6.31	8.541
TId	11.00 (63%)	19.655 (33.7%)	27.73 (7.6%)	27.682 (6.6%)	30.03	29.647
memplus	44.21 (87%)	68.467 (50%)	338.22 (1.8%)	138.633 (-0.3%)	344.64	138.218
TIb	73.97 (94%)	107.711 (68%)	1213.47 (8%)	315.412 (7%)	1325.29	341.418
wang3	645.15 (55%)	347.505 (19%)	1417.70 (1%)	428.741 (0.5%)	1431.91	430.817

5 Experimental Studies on Parallel Performance

Our experiments on Cray T3E show that the parallel time performance of *LazyS*$^+$ is still competitive to S^+. It is shown in Table 5 that *LazyS*$^+$ can achieve 10.004 GFLOPS on matrix vavasis, which is not much less than the highest 11.04 GFLOPS achieved by S^+ on 128 450MHz T3E nodes. Table 6 is the performance on 300Mhz T3E nodes. Our study focuses on relatively large matrices.

TABLE 5

Time and MFLOPS performance of $LazyS^+$ and S^+ on 450MHz Cray T3E. A "-" implies the data is not available due to insufficient memory. A "" implies the data is not available due to insufficient CPU quota on this machine. Time is in seconds.*

Matrix	$LazyS^+$ P=8		S^+ P=8		$LazyS^+$ P=128		S^+ P=128	
	Time	MFLOPS	Time	MFLOPS	Time	MFLOPS	Time	MFLOPS
goodwin	*	*	1.21	553.5	*	*	0.67	999.6
e40r0100	*	*	4.06	611.3	*	*	1.59	1560.9
raefsky4	*	*	38.62	804.2	*	*	4.55	6826.0
af23560	*	*	10.57	602.1	*	*	2.80	2272.9
vavasis3	59.77	1492.9	62.68	1423.6	8.92	10004.0	8.08	11043.5
TIa	0.61	339.6	0.64	323.7	0.28	739.9	0.26	796.8
TId	2.10	281.5	1.98	298.6	0.59	1001.9	0.54	1094.8
TIb	12.81	555.7	47.88	148.7	2.83	2515.7	4.98	1429.5
wang3	74.69	194.9	-	-	9.04	1610.2	-	-

Table 5 lists the parallel performance on 450MHz T3E nodes, the performance data of $LazyS^+$ for some non circuit simulation matrices is not available due to the insufficient CPU quota on this machine [1]. Nevertheless, data on 300MHz T3E nodes in Table 5 actually indicates that $LazyS^+$ is competitive with S^+ for these matrices. For the matrices with high fill-in overestimation ratios, we observe that $LazyS^+$ with dynamic space management is better than S^+. It is about 191% faster on 8 processors and 120% faster on 128 processors. Matrix wang3 can't run on T3E using S^+ since they produce too many fill-ins from static symbolic factorization. However $LazyS^+$ only allocates space if necessary, so considerable space is saved for a large amount of zero blocks.

As for other matrices, we can see from Table 6 that on 8 300Mhz processors $LazyS^+$ is about 1% slower than S^+ while on 128 processors, $LazyS^+$ is 7% slower than S^+. On average, $LazyS^+$ tends to become slower when the number of processors becomes larger. This is because the lazy allocation scheme introduces new overhead for dynamic memory management and for row and column broadcasts (blocks of the same L-column or U-row, now allocated in non-contiguous memory, can no longer be broadcasted as a unit). This new overhead affects critical paths, which dominate performance when parallelism is limited and the number of processors is large. This problem tends to become more serious when the number of processors is getting bigger.

6 Concluding Remarks

The proposed space optimization techniques effectively reduce memory requirements when static symbolic factorization creates an excessive amount of extra fill-ins. The new algorithm with dynamic space management exhibits competitive sequential space and time performance compared to SuperLU for the tested matrices. The parallel code becomes more robust in handling different classes of sparse matrices. In the future, it is interesting to study impact of matrix ordering and compare the other approaches that handle nonsymmetric matrices using as the multifrontal method [1] and static pivoting [15].

[1] We will provide it when resource is available.

Table 6

MFLOPS performance of S^+ and $LazyS^+$ on 300MHz Cray T3E.

Matrix	P=8		P=32		P=128	
	$LazyS^+$	S^+	$LazyS^+$	S^+	$LazyS^+$	S^+
goodwin	374.1	403.5	676.4	736.0	788.0	826.8
e40r0100	413.0	443.2	880.2	992.8	1182.0	1272.8
raefsky4	587.6	568.2	1922.1	1930.3	4875.8	5133.6
af23560	418.4	432.1	1048.4	1161.3	1590.9	1844.7
vavasis3	1031.7	958.4	3469.4	3303.6	7924.5	8441.9

Acknowledgment. This work was supported in part by NSF CAREER CCR-9702640 and by DARPA through UMD (ONR Contract Number N6600197C8534). We would like to thank Horst Simon for providing access to the Cray 450Mhz T3E at NERSC.

References

[1] P. R. Amestoy, I. S. Duff, and J.-Y. L'Execellent. Multifrontal parallel distributed symmetric and unsymmetric solvers. Technical Report RAL-TR-98-051, Rutherford Appleton Laboratory, 1998.

[2] T. Davis and I. S. Duff. An Unsymmetric-pattern Multifrontal Method for Sparse LU factorization. *SIAM Matrix Analysis & Applications*, January 1997.

[3] J. Demmel. Numerical Linear Algebra on Parallel Processors. Lecture Notes for NSF-CBMS Regional Conference in the Mathematical Sciences, June 1995.

[4] J. Demmel, S. Eisenstat, J. Gilbert, X. Li, and J. Liu. A Supernodal Approach to Sparse Partial Pivoting. Technical Report CSD-95-883, EECS Department, UC Berkeley, September 1995. To appear in SIAM J. Matrix Anal. Appl.

[5] J. Demmel, J. Gilbert, and X. Li. An Asynchronous Parallel Supernodal Algorithm for Sparse Gaussian Elimination. Technical Report CSD-97-943, EECS Department, UC Berkeley, February 1997. To appear in SIAM J. Matrix Anal. Appl.

[6] J. Dongarra, J. Du Croz, S. Hammarling, and R. Hanson. An Extended Set of Basic Linear Algebra Subroutines. *ACM Trans. on Mathematical Software*, 14:18–32, 1988.

[7] I. S. Duff. On Algorithms for Obtaining a Maximum Transversal. *ACM Transactions on Mathematical Software*, 7(3):315–330, September 1981.

[8] C. Fu, X. Jiao, and T. Yang. A Comparison of 1-D and 2-D Data Mapping for Sparse LU Factorization on Distributed Memory Machines. *Proc. of 8th SIAM Conference on Parallel Processing for Scientific Computing*, March 1997.

[9] C. Fu, X. Jiao, and T. Yang. Efficient Sparse LU Factorization with Partial Pivoting on Distributed Memory Architectures. *IEEE Transactions on Parallel and Distributed Systems*, 9(2):109–125, February 1998.

[10] C. Fu and T. Yang. Sparse LU Factorization with Partial Pivoting on Distributed Memory Machines. In *Proceedings of ACM/IEEE Supercomputing*, Pittsburgh, November 1996.

[11] C. Fu and T. Yang. Space and Time Efficient Execution of Parallel Irregular Computations. In *Proceedings of ACM Symposium on Principles & Practice of Parallel Programming*, June 1997.

[12] A. George and E. Ng. Parallel Sparse Gaussian Elimination with Partial Pivoting. *Annals of Operations Research*, 22:219–240, 1990.

[13] X. Jiao. Parallel Sparse Gaussian Elimination with Partial Pivoting and 2-D Data Mapping. Master's thesis, Dept. of Computer Science, University of California at Santa Barbara, August 1997.

[14] X. Li. *Sparse Gaussian Elimination on High Performance Computers.* PhD thesis, Computer Science Division, EECS, UC Berkeley, 1996.

[15] X. S. Li and J. W. Demmel. Making Sparse Gaussian Elimination Scalable by Static Pivoting. In *Proceedings of Supercomputing'98*, 1998.

[16] K. Shen, X. Jiao, and T. Yang. Elimination Forest Guided 2D Sparse LU Factorization. In *Proceedings of the 10th ACM Symposium on Parallel Algorithms and Architectures*, pages 5–15, June 1998.