S^+: EFFICIENT 2D SPARSE LU FACTORIZATION ON PARALLEL MACHINES*

KAI SHEN†, TAO YANG‡, AND XIANGMIN JIAO§

Abstract. Static symbolic factorization coupled with supernode partitioning and asynchronous computation scheduling can achieve high gigaflop rates for parallel sparse LU factorization with partial pivoting. This paper studies properties of elimination forests and uses them to optimize supernode partitioning/amalgamation and execution scheduling. It also proposes supernodal matrix multiplication to speed-up kernel computation by retaining the BLAS-3 level efficiency and avoiding unnecessary arithmetic operations. The experiments show that our new design with proper space optimization, called S^+, improves our previous solution substantially and can achieve up to 10 GFLOPS on 128 Cray T3E 450MHz nodes.

1. Introduction. The solution of sparse linear systems is a computational bottleneck in many scientific computing problems. When dynamic pivoting is required to maintain numerical stability in direct methods for solving non-symmetric linear systems, it is challenging to develop high performance parallel code because pivoting causes severe caching miss and load imbalance on modern architectures with memory hierarchies. The previous work has addressed parallelization on shared memory platforms or with restricted pivoting [4, 13, 15, 19]. Most notably, the recent shared memory implementation of SuperLU has achieved up to 2.58GFLOPS on 8 Cray C90 nodes [4, 5, 23]. For distributed memory machines, we proposed an approach that adopts a static symbolic factorization scheme to avoid data structure variation [10, 11]. Static symbolic factorization eliminates the runtime overhead of dynamic symbolic factorization with a price of over-estimated fill-ins and thereafter extra computation [15]. However, the static data structure allowed us to identify data regularity, maximize the use of BLAS-3 operations, and utilize task graph scheduling techniques and efficient run-time support [12] to achieve high efficiency.

This paper addresses three issues to further improve the performance of parallel sparse LU factorization with partial pivoting on distributed memory machines. First, we study the use of elimination trees in optimizing matrix partitioning and task scheduling. Elimination trees or forests are used extensively in sparse Cholesky factorization [18, 26, 27] because they have a more compact representation of parallelism than task graphs. For sparse LU factorization, the traditional approach uses the elimination tree of A^TA, which can produce excessive false computational dependency. In this paper, we use elimination trees(forest) of A to guide matrix partitioning and parallelism control in LU factorization. We show that improved supernode partitioning and amalgamation effectively control extra fill-ins and produce optimized supernodal partitioning. We also use elimination forests to identify data dependence and potential concurrency among pivoting and updating tasks and thus maximize utilization of limited parallelism.

Second, we propose a fast and space-efficient kernel for supernode-based matrix

*This work was supported in part by NSF CCR-9702640 and by DARPA through UMD (ONR Contract Number N6600197C8534).

†Department of Computer Science, University of California at Santa Barbara, CA 93106, USA (kshen@cs.ucsb.edu)

‡Department of Computer Science, University of California at Santa Barbara, CA 93106, USA (tyang@cs.ucsb.edu)

§Department of Computer Science, University of Illinois at Urbana-Champaign, IL 61801, USA (jiao@cs.uiuc.edu)

multiplication to improve the performance of sparse LU factorization. This is based on the observation that nonzero submatrices generated by supernodal partitioning and amalgamation have special patterns. Namely, they contain either dense subrows or subcolumns. This new kernel avoids unnecessary arithmetic operations while retains the BLAS-3 level efficiency.

Third, we evaluate space requirement of static factorization and propose an optimization scheme which acquires memory on-the-fly only when it is necessary. This scheme can effectively control peak memory usage, especially when static symbolic factorization overestimates fill-ins excessively.

Our new design with these optimizations, called S^+, improves our previous code by more than 50% in execution time. In particular S^+ without space optimization achieved up to 8.25 GFLOPS on 128 T3E 300MHz nodes and 10.85 GFLOPS[1] on 128 T3E 450MHz nodes. The space optimization technique slightly degrades overall time efficiency but it reduces space requirement by up to 68% in some cases. S^+ with space optimization can still deliver up to 10.00GFLOPS on 128 Cray 450Mhz T3E nodes. Notice that we only count *true* operations, in the sense that no extra arithmetic operation introduced by static factorization or amalgamation is included in computing gigaflop rates of our algorithm.

The rest of this paper is organized as follows. Section 2 gives the background knowledge for sparse LU factorization. Section 3 presents a modified definition and properties of elimination trees for sparse LU factorization, and their applications in supernode partitioning and amalgamation. Section 4 describes our strategies of exploiting 2D asynchronous parallelism. Section 5 discusses a fast matrix multiplication kernel suitable for submatrices derived from supernode partitioning. Section 6 presents experimental results on Cray T3E. Section 7 discusses space optimization for S^+. Section 8 concludes the paper. A summary of notations and the proof for each theorem are listed in the appendix.

2. Background. LU factorization with partial pivoting decomposes a non-symmetric sparse matrix A into two matrices L and U, such that $PA = LU$, where L is a unit lower triangular matrix, U is an upper triangular matrix, and P is a permutation matrix containing pivoting information.

Static symbolic factorization. A static symbolic factorization approach is proposed in [14] to identify the worst case nonzero patterns for sparse LU factorization without knowing numerical values of elements. The basic idea is to statically consider all possible pivoting choices at each elimination step and space is allocated for all possible nonzero entries. Static symbolic factorization annihilates data structure variation, and hence it improves predictability of resource requirements and enables static optimization strategies. On the other hand, dynamic factorization, which is used in SuperLU [4, 23], provides more accurate control of data structures on the fly. But it is challenging to parallelize dynamic factorization with low runtime overhead on distributed memory machines.

The static symbolic factorization for an $n \times n$ matrix is outlined as follows. At each step $k (1 \leq k < n)$, each row $i \geq k$ which has a nonzero element in column k is a *candidate pivot row* for row k. As the static symbolic factorization proceeds, at step k the nonzero structure of *each* candidate pivot row is replaced by the union of the structures of all these candidate pivot rows except the elements in the first

[1] We reported a performance record of 11.04 GFLOPS in an earlier paper [29]. We later found that the operation count included extra computation due to amalgamation. In this paper, we disabled amalgamation in operation counting.

$k-1$ columns. Using an efficient implementation [21] for the symbolic factorization algorithm proposed in [14], this preprocessing step can be very fast. For example, it costs less than one second for most of our test matrices, and at worst it costs 2 seconds on a single node of Cray T3E. The memory requirement is also fairly small. If LU factorization is used in an iterative numerical method, then the cost of symbolic factorization together with other preprocessing is amortized over multiple iterations.

In the previous work, we show that static factorization does not produce too many fill-ins for most of our test matrices, even for large matrices using a simple matrix ordering strategy (minimum degree ordering) [10, 11]. For a few matrices that we have tested, static factorization generates an excessive number of fill-ins. In Section 7, we discuss space optimization for S^+ in addressing such a problem.

L/U **supernode partitioning.** After the fill-in pattern of a matrix is predicted, the matrix is further partitioned using a supernodal approach to improve caching performance. In [23], a non-symmetric supernode is defined as a group of consecutive columns, in which the corresponding L part 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 scheme L *supernode partitioning.* Here by "subrow", we mean the contiguous part of a row within a supernode.

After an L supernode partitioning has been performed on a sparse matrix A, the same partitioning is applied to the rows of A to further break each supernode into submatrices. This is also known as U *supernode partitioning.* Since coarse-grain partitioning can reduce available parallelism and produce large submatrices which do not fit into the cache, an upper bound on the supernode size is usually enforced in the L/U supernode partitioning. 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 part of A contains only dense subcolumns [11]. This is the key to maximize the use of BLAS-3 subroutines [7] 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 [7]. Figure 1 illustrates an example of a partitioned sparse matrix and the black areas depict dense submatrices, subrows, and subcolumns.

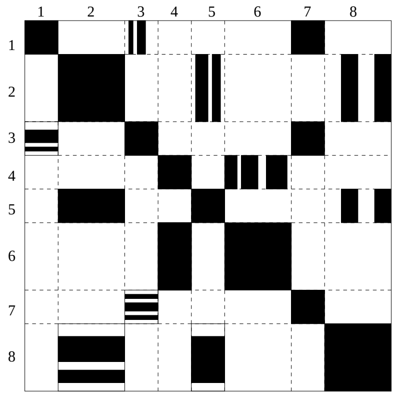

FIG. 1. *Example of a partitioned sparse matrix.*

Data mapping. After symbolic factorization and matrix partitioning, a partitioned sparse matrix A has $N \times N$ submatrix blocks. For example, the matrix in Figure 1 has 8×8 submatrices. Let $A_{i,j}$ denote the submatrix in A with row block index i and column block index j. Let $L_{i,j}$ and $U_{i,j}$ denote a submatrix in the lower and upper triangular part of matrix A respectively. For block-oriented matrix computation, 1D column block cyclic mapping and 2D block cyclic mapping are commonly used. In 1D column block cyclic mapping, a column block of A is assigned to one processor. In 2D mapping, processors are viewed as a 2D grid, and a column block is assigned to a column of processors. 2D sparse LU factorization is more scalable than 1D data mapping [10]. However, 2D mapping introduces more overhead for pivoting and row swapping. Since asynchronous execution requires extensive use of buffers, in designing 2D codes, we need to pay special attention to the usage of buffer space, so that our 2D code is able to factorize larger matrices under memory constraints.

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

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

Program partitioning. Each column block k is associated with two types of tasks: $Factor(k)$ and $Update(k, j)$ for $1 \leq k < j \leq N$. Task $Factor(k)$ factorizes all the columns in the kth column block and its function includes 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 kth column block is completed. Then the pivoting sequence is applied to the rest of the matrix. This is called "delayed pivoting" [3]. Task $Update(k, j)$ uses column block k $(L_{k,k}, L_{k+1,k}, \cdots, L_{N,k})$ to modify column block j. That includes "row swapping" which applies the pivoting derived by $Factor(k)$ to column block j, "scaling" that uses the factorized submatrix $L_{k,k}$ to scale $U_{k,j}$, and "updating" that uses submatrices $L_{i,k}$ and $U_{k,j}$ to modify $A_{i,j}$ for $k + 1 \leq i \leq N$. Figure 2 outlines the partitioned LU factorization algorithm with partial pivoting.

3. Elimination forests and non-symmetric supernode partitioning. In this section, we study properties of elimination forests [1, 15, 16, 25][2] and use them to design more robust strategies for supernode partitioning and parallelism detection. As a result, both sequential and parallel versions of our code can be improved.

We will use the following notations in our discussion. Let A be the given $n \times n$ sparse matrix. Notice that the nonzero structure of matrix A changes after symbolic factorization and the algorithm design discussed in the rest of this paper addresses A after symbolic factorization. Let $a_{i,j}$ be the element in A with row index i and column index j, and $a_{i:j,s:t}$ be the submatrix in A from row i to row j and from column s to t. Let l_k be column k in the lower triangular part and let u_k be row k in the upper triangular part of A after symbolic factorization. Notice that both l_k and u_k include $a_{k,k}$. To emphasize nonzero patterns of A, we use symbol ˆ to express the nonzero structure after symbolic factorization. Expression $\hat{a}_{i,j} \neq 0$ means that $a_{i,j}$

[2]An elimination forest only has one tree when the corresponding sparse matrix is irreducible. In that case, it is also called an elimination tree.

is not zero after symbolic factorization. We assume that every diagonal element in the original sparse matrix 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 A to obtain a matrix with zero-free diagonal [8]. Let $\hat{l}_k$ be the index set of nonzeros in l_k, i.e. $\{i \mid \hat{a}_{i,k}\neq 0 \wedge i\geq k\}$. Similarly, let $\hat{u}_k$ be the index set of nonzeros in u_k, i.e. $\{j \mid \hat{a}_{k,j}\neq 0 \wedge j\geq k\}$. Symbol $|\hat{l}_k|$ (or $|\hat{u}_k|$) denotes the cardinality of $\hat{l}_k$ (or $\hat{u}_k$).

3.1. The definition of elimination forests. We study the elimination forest of a matrix which may or may not be reducible. Previous research on elimination forests has shown that an elimination forest contains information about all potential dependency if the corresponding sparse matrix is irreducible [1, 15, 16, 25]. Although it is always possible to decompose a reducible matrix into several smaller irreducible matrices, the decomposition introduces extra burden on software design and implementation. Instead, we generalize the original definition of elimination tree to reducible matrices. Our definition, listed in Definition 3.1, differs from the original definition by imposing condition $|\hat{l}_k| > 1$. Imposing this condition not only avoids some false dependency, but also allows us to generalize the some results for irreducible matrices to reducible matrices, which are summarized in Theorems 3.2 and 3.4. Note that when A is irreducible, the condition $|\hat{l}_k| > 1$ holds for all $1 \leq k < n$ and the new definition generates the same elimination forest as the original definition. In practice, we find that some test matrices can have up to 50% of columns with zero lower-diagonal nonzeros after symbolic factorization.

DEFINITION 3.1. *An* **LU Elimination forest** *for an $n \times n$ matrix A has n vertices numbered from 1 to n. For any two numbers k and j ($k < j$), there is an edge from vertex j to vertex k in the forest if $a_{k,j}$ is the first off-diagonal nonzero in $\hat{u}_k$ and $|\hat{l}_k| > 1$. Vertex j is called the* **parent** *of vertex k, and vertex k is called a* **child** *of vertex j.*

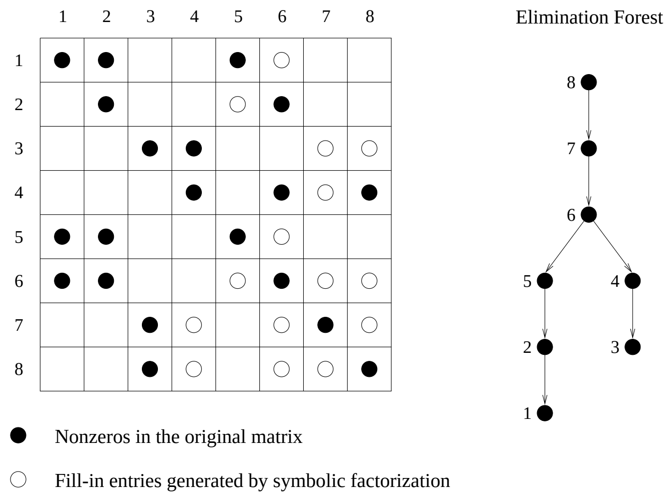

FIG. 3. *A sparse matrix and its elimination forest.*

An elimination forest for a given matrix can be generated in a time complexity of $O(n)$ if computed as a byproduct of symbolic factorization. Figure 3 illustrates a sparse matrix after symbolic factorization and its elimination forest. We now discuss two properties of an elimination forest for a general sparse matrix.

THEOREM 3.2. *If vertex j is an ancestor of vertex k in the elimination forest, then* $\{r \mid r{\in}\hat{l}_k \wedge j{\leq}r{\leq}n\} \subseteq \hat{l}_j$, *and* $\{c \mid c{\in}\hat{u}_k \wedge j{\leq}c{\leq}n\} \subseteq \hat{u}_j$.

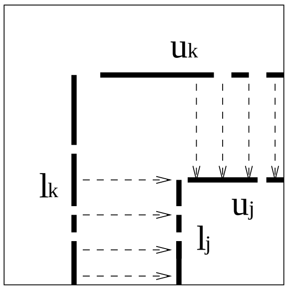

FIG. 4. *An illustration of Theorem 3.2 (vertex j is an ancestor of vertex k in the elimination forest).*

Theorem 3.2 (illustrated in Figure 4) captures the structural containment between two columns in L and between two rows in U. It indicates that the nonzero structure of l_j (or u_j) subsumes l_k (or u_k) if the corresponding vertices have an ancestor relationship. This information will be used for designing supernode partitioning with amalgamation in the next subsection.

DEFINITION 3.3. *Let $j > k$, l_k* **directly updates** *l_j if task Update(k, j) is performed in LU factorization, i.e. $\hat{a}_{k,j} \neq 0$ and $|\hat{l}_k| > 1$. l_k* **indirectly updates** *l_j if there is a sequence $s_1, s_2, \cdots, s_p$ such that: $s_1 = k$, $s_p = j$ and l_{s_q} directly updates $l_{s_{q+1}}$ for each $1 \leq q \leq p-1$.*

THEOREM 3.4. *Let $k < j$, l_k directly or indirectly updates l_j in LU factorization if and only if vertex j is an ancestor of vertex k in the elimination forest.* Theorem 3.4 indicates dependency information during numerical factorization, which can guide the scheduling of asynchronous parallelism.

3.2. 2D L/U supernode partitioning and amalgamation. Given a nonsymmetric matrix A after symbolic factorization, in [11] we have described a two-stage L/U supernode partitioning method: At Stage 1, a group of consecutive columns that have the same structure in the L part is considered as one supernode column block. Then the L part is sliced as a set of consecutive column blocks. After an L supernode partition has been obtained, at Stage 2 the same partition is applied to rows of the matrix to break each supernode column block further into submatrices.

We examine how elimination forests can be used to guide and improve the 2D L/U supernode partitioning. The following corollary is a straightforward result of Theorem 3.2 and it shows that we can easily traverse an elimination forest to identify supernodes. Notice that each element in a dense structure can be a nonzero or a fill-in due to static symbolic factorization.

COROLLARY 3.5. *If for each $k \in \{s+1, s+2, \cdots, t\}$, vertex k is the parent of vertex $k-1$ and $|\hat{l}_k| = |\hat{l}_{k-1}| - 1$, then after symbolic factorization, 1) diagonal block $a_{s:t,\ s:t}$ is completely dense, 2) $a_{t+1:n,s:t}$ contains only dense subrows, and 3) $a_{s:t,t+1:n}$ contains only dense subcolumns.*

The partitioning algorithm using the above corollary is briefly summarized as follows. For each pair of two consecutively numbered vertices with the parent/child relationship in the elimination forest, we check the size difference between the two corresponding columns in the L part. If the difference is one, we assign these two columns into an L supernode. Since if a submatrix in a supernode is too large, it won't fit into the cache and also large grain partitioning reduces available parallelism, we usually enforce an upper bound on the supernode size. Notice that U partitioning is applied after the L partitioning is completed. We need not check any constraint on U because as long as a child-parent pair $(i, i-1)$ satisfies $|\hat{l}_i| = |\hat{l}_{i-1}| - 1$, it also satisfies $|\hat{u}_i| = |\hat{u}_{i-1}| - 1$ due to Theorem 1 in [10, 11]. Hence the structures of u_i and u_{i-1} are identical. Figure 5(a) illustrates supernode partitioning of the sparse matrix in Figure 3. There are 6 L/U supernodes in this figure. From the L partitioning point of view, columns from 1 to 5 are not grouped but columns 6, 7 and 8 are clustered together.

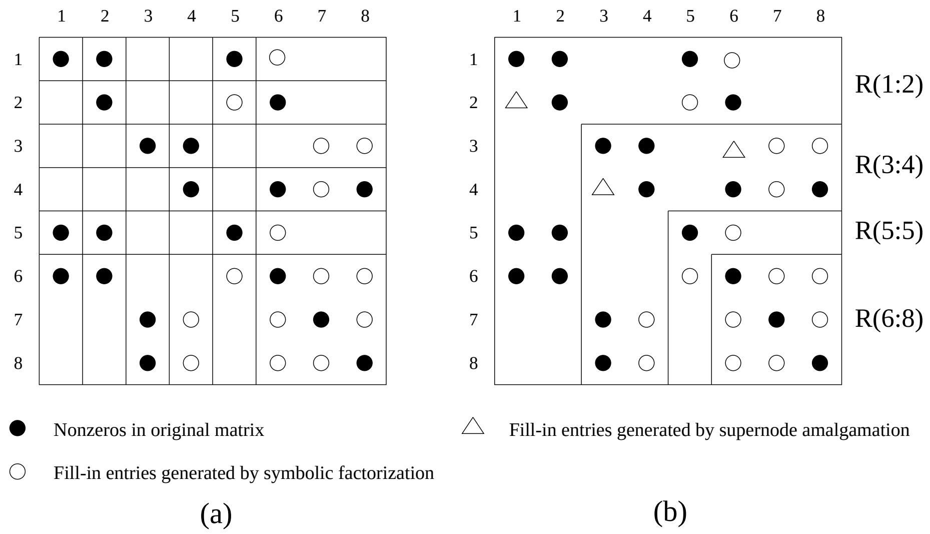

FIG. 5. *(a) Supernode partitioning for the matrix in Figure 3; (b) The result of supernode amalgamation with 4 related L/U supernodes.*

For most of the test matrices in our experiments, the average supernode size after the above partitioning strategy is very small, about 1.5 to 2 columns. This leads to relatively fine grained computation. In practice, amalgamation is commonly adopted to increase the average supernode size by introducing some extra zero entries in the dense structures of supernodes. In this way, caching performance can be improved and interprocessor communication overhead may be reduced. For sparse Cholesky factorization(e.g. [26]), the basic idea of amalgamation is to relax the restriction that all the columns in a supernode must have exactly the same off-diagonal nonzero structure. In a Cholesky elimination tree, a parent could be merged with its children if merging does not introduce too many extra zero entries into a supernode. Row and column permutations are needed if the parent is not consecutive with its children. For sparse LU factorization, such a permutation may alter the result of symbolic factorization. In our previous approach [11], we simply compare consecutive columns of the L part, and make a decision on merging if the total number of difference is under a pre-set threshold. This approach is simple, resulting in a bounded number of extra zero entries included in the dense structure of an L supernode. However, the result of partitioning may lead to too many extra zero entries in the dense structure of a U supernode. Using Theorem 3.2, we can remedy this problem as follows by partitioning L and U parts simultaneously and controlling the number of fill-ins in both L and U.

We consider a supernode containing elements from both L and U parts, and refer to a supernode after amalgamation as a *relaxed L/U supernode.* The definition is listed below.

DEFINITION 3.6. *A* **relaxed L/U supernode** *$R(s{:}t)$ contains three parts: the diagonal block $a_{s:t,s:t}$, the L supernode part $a_{t+1:n,s:t}$ and the U supernode part $a_{s:t,t+1:n}$. The* **supernode size** *of $R(s:t)$ is $t-s+1$.*

A partitioning example illustrated in Figure 5(b) has four relaxed L/U supernodes: $R(1:2)$, $R(3:4)$, $R(5:5)$, and $R(6:8)$. The following corollary, which is also a straightforward result of Theorem 3.2, can be used to bound the nonzero structure of a relaxed L/U supernode.

COROLLARY 3.7. *If for each k where $s+1 \leq k \leq t$, vertex k is the parent of vertex $k-1$ in an elimination forest, then $\{i \mid i \in \hat{l}_k \wedge t \leq i \leq n\} \subseteq \hat{l}_t$, and $\{j \mid j \in \hat{u}_k \wedge t \leq j \leq n\} \subseteq \hat{u}_t$.*

Using Corollary 3.7, in $R(s:t)$ the ratio of extra fill-ins introduced by amalgamation compared with the actual nonzeros can be computed as:

$$z = \frac{(t-s+1)^2 + (t-s+1) \times (|\hat{l}_t| + |\hat{u}_t| - 2)}{nz(R(s:t))} - 1$$

where $nz()$ gives the number of nonzero elements in the corresponding structure including fill-ins created by symbolic factorization. Also notice that both $\hat{l}_t$ and $\hat{u}_t$ include diagonal element $a_{t,t}$.

Thus our heuristic for 2D partitioning is to traverse the elimination forest and find relaxed supernodes $R(s:t)$ satisfying the following conditions:

1. for each i where $s+1 \leq i \leq t$, vertex i is the parent of vertex $i-1$ in the elimination forest,
2. the extra fill-in ratio, z, is less than the pre-defined threshold, and
3. $t-s+1 \leq$ the pre-defined upper bound for supernode sizes.

The complexity of such a partitioning algorithm with amalgamation is $O(n)$, which is very low and is made possible by Corollary 3.7. Our experiments show that the above strategy is very effective. The number of total extra fill-ins doesn't change much when the upper bound for z is in the range of $10-100\%$ and it seldom exceeds 2% of the total nonzeros in the whole matrix. In terms of upper bound for supernode size, 25 gives the best caching and parallel performance on the T3E. Thus all the experiments in Section 6 are completed with $z \leq 30\%$ and *supernode size* ≤ 25. Figure 5(b) is the result of supernode amalgamation for the sparse matrix in Figure 3 using condition $z \leq 30\%$.

In the rest of this paper, we will call relaxed L/U supernodes simply as supernodes.

Compressed storage scheme for submatrices. In our implementation, every submatrix is stored in a compressed storage scheme with a bit map to indicate its nonzero structure. In addition to the storage saving, the compressed storage scheme can also eliminate certain unnecessary computations on zero elements which will be discussed in details in Section 5. For an L submatrix, its subrows are stored in a consecutive space even though their corresponding row numbers may not be consecutive. The bit map is used to identify dense subrows in L submatrices. A bit is set to 0 if the corresponding subrow is zero, and set to 1 otherwise. Figure 6 illustrates such a storage scheme for a 6×8 L submatrix. In this example, the second, third and fifth subrows are dense and all other subrows are completely zero. The strategy for a U

submatrix is the same except in a subcolumn-oriented fashion. Since level-1 cache is not large in practice and the supernode size is limited to fit the cache (limit is 25 on Cray T3E), we can use a 32-bit word to store the bit map of each submatrix, and can determine if a subrow is dense efficiently using a single logical “and” operation.

Space overhead for a submatrix includes the bit map and global matrix index. Index information is piggybacked on a message when sending submatrices among processors. In terms of space for bit maps, if a submatrix is completely zero, its bit map vector is not needed. For a non-zero submatrix, the size of its bit map is just one word. Thus numerical values of the sparse matrix always dominate the overall storage requirement and space overhead for bit map vectors is insignificant. It should also be noted that in a future CPU architecture with a large level-1 cache, a 32-bit word may not be sufficient and some minor changes in the implementation are needed to use two words or more. In this case, using more than one word for a bit vector should not cause space concern because amalgamation ensures the average submatrix size is not too small. Also this compression scheme can be turned off for an extremely small submatrix (but we do not expect such a thing is needed in practice).

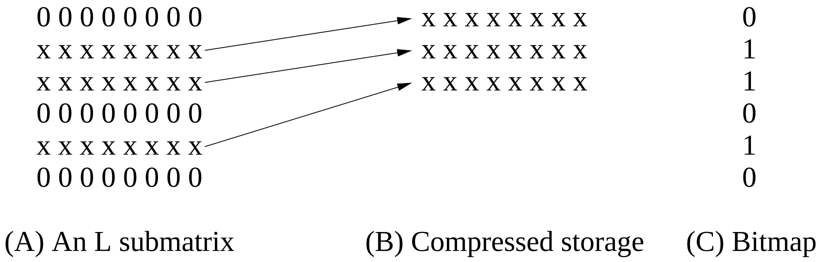

FIG. 6. *An illustration of a compressed storage scheme for an L submatrix.*

4. 2D asynchronous parallelism exploitation. In this section, we present scheduling strategies for exploiting asynchronous 2D parallelism so that different updating stages can be overlapped. After 2D L/U supernode partitioning and amalgamation, the $n \times n$ sparse matrix A is 2-dimensionally partitioned into $N \times N$ submatrices. Let symbol $A_{i,j}$ denote the submatrix in row block i and column block j and let $A_{i:j,s:t}$ denote all submatrices from row block i to j and from column block s to t. Let $L_{i,j}$ and $U_{i,j}$ $(i \neq j)$ denote submatrices in the lower and upper triangular part respectively. Our 2D algorithm uses the standard cyclic mapping since it tends to distribute data evenly, which is important to solve large problems. In this scheme, p available processors are viewed as a two dimensional grid: $p = p_r \times p_c$. Then block $A_{i,j}$ is assigned to processor $P_{i \bmod p_r,\, j \bmod p_c}$.

In Section 2, we have described two types of tasks involved in LU factorization. One is $Factor(k)$, which is to factorize all the columns in the kth column block, including finding the pivoting sequence associated with those columns. The other is $Update(k, j)$, which is to apply the pivoting sequence derived from $Factor(k)$ to the jth column block, and modify the jth column block using the kth column block, where $k < j$ and $U_{k,j} \neq 0$. The 2D data mapping enables parallelization of a single $Factor(k)$ or $Update(k, j)$ task on p_r processors because each column block is distributed into p_r processors. The main challenge is the coordination of pivoting and data swapping across a subset of processors to exploit as much parallelism as possible with low buffer space demand.

For task $Factor(k)$, the computation is distributed among processors in column

$k \bmod p_c$ of the processor grid, and global synchronization among this processor column is needed for correct pivoting. To simplify the parallelism control of task $Update(k, j)$ we split it into two subtasks: $ScaleSwap(k)$ which does scaling and delayed row interchange for submatrices $A_{k:N,\ k+1:N}$, and $Update2D(k, j)$ which modifies column block j using column block k. For $ScaleSwap(k)$, the synchronization among processors within the same column of the grid is needed. For $Update2D(k, j)$, no synchronization among processors is needed as long as the desired submatrices in column blocks k and j are made available to processor $P_{i \bmod p_r,\ j \bmod p_c}$ where $k+1 \leq i \leq N$.

We discuss three scheduling strategies below. The first one as reported in [9] is a basic approach in which computation flow is controlled by pivoting tasks $Factor(k)$. The order of execution for $Factor(k)$, $k = 1, 2, \cdots, N$ is sequential, but $Update2D()$ tasks, where most of the computation comes from, can execute in parallel among all processors. Let symbol $Update2D(k, *)$ denote tasks $Update2D(k, t)$ for $k+1 \leq t \leq N$. The asynchronous parallelism comes from two levels. First a single stage of tasks $Update2D(k, *)$ can be executed concurrently on all processors. In addition, different stages of $Update2D()$ tasks from $Update2D(k, *)$ and $Update2D(k', *)$, where $k \neq k'$, can also be overlapped.

The second approach is called factor-ahead which improves the first approach by letting $Factor(k+1)$ start as soon as $Update2D(k, k+1)$ completes. This is based on an observation that in the basic approach, after all tasks $Update2D(k, *)$ are done, all processors must wait for the result of $Factor(k+1)$ to start $Update2D(k+1, *)$. It is not necessary that $Factor(k+1)$ has to wait for the completion of all tasks $Update2D(k, *)$. This idea has been used in the dense LU algorithm [17] and we extend it for asynchronous execution and incorporate a buffer space control mechanism. The details are in [10].

The factor-ahead technique still imposes a constraint that $Factor(k+1)$ must be executed after the completion of $Factor(k)$. In order to exploit potential parallelism between $Factor()$ tasks, our third design is to utilize dependence information represented by elimination forests. Since we deal with a partitioned matrix, the elimination forest defined in Definition 3.1 needs to be clustered into a supernode-wise elimination forest. We call the new forest as a **supernodal elimination forest**. And we call the element-wise elimination forest as a **nodal elimination forest**.

DEFINITION 4.1. *A supernodal elimination forest has N nodes. Each node corresponds to a relaxed L/U supernode. Supernode $R(i_1 : i_2)$ is the parent of supernode $R(j_1 : j_2)$ if there exists vertex $i \in \{i_1, i_1+1, \cdots, i_2\}$ and vertex $j \in \{j_1, j_1+1, \cdots, j_2\}$ such that i is j's parent in the corresponding nodal elimination forest.*

A supernodal elimination forest can be generated efficiently in $O(n)$ time using Theorem 4.2 listed below. Figure 7 illustrates the supernodal elimination forest for Figure 5(b). The corresponding matrix is partitioned into 4×4 submatrices.

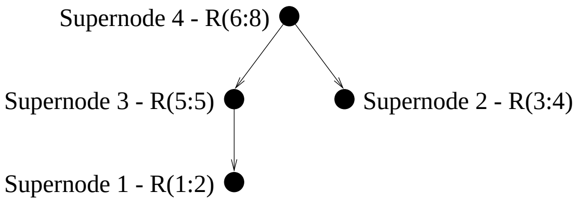

FIG. 7. *Supernodal elimination forest for the matrix in Figure 5(b)*

THEOREM 4.2. *Supernode $R(i_1 : i_2)$ is the parent of supernode $R(j_1 : j_2)$ in the supernodal elimination forest if and only if there exists vertex $i \in \{i_1, i_1+1, \cdots, i_2\}$ which is the parent of vertex j_2 in the nodal elimination forest.*

Finally the following theorem indicates computation dependence among supernodes and exposes the possible parallelism that can be exploited.

THEOREM 4.3. *The L part of supernode $R(j_1 : j_2)$ directly or indirectly updates the L part of supernode $R(i_1 : i_2)$ if and only if $R(i_1 : i_2)$ is an ancestor of supernode $R(j_1 : j_2)$.*

Our design for LU factorization task scheduling using the above forest concept is different from the ones for Cholesky factorization [1, 26] because pivoting and row interchanges complicate the flow control in LU factorization. Using Theorem 4.3, we are able to exploit some parallelism among $Factor()$ tasks. After tasks $Factor(i)$ and $Update2D(i, k)$ have finished for every child i of supernode k, task $Factor(k)$ is ready for execution. Because of the space constraint on the buffer size, our current design does not fully exploit the parallelism and this design is explained below.

Space complexity. We examine the degree of parallelism exploited in our algorithm by determining the maximum number of updating stages that can be overlapped. Using this information we can estimate the extra buffer space needed per processor for asynchronous execution. This buffer is used to accommodate nonzeros in $A_{k:N,k}$ and the pivoting sequence at each elimination step k. We define the **stage overlapping degree** for updating tasks as

$$max\{|k - k'| \;\big|\; Update2D(k, *) \text{ and } Update2D(k', *) \text{ can execute concurrently.}\}$$

It is shown in [10] that for the factor-ahead approach, the reachable overlapping degree is p_c among all processors and the extra buffer space complexity is about $\frac{2.5 \cdot BSIZE}{n} \cdot S_1$ where S_1 is the sequential space size for storing the entire sparse matrix and $BSIZE$ is the maximum supernode size. This complexity is very small for a large matrix. Also because 2D cyclic mapping normally leads to a uniform data distribution, our factor-head approach is able to handle large matrices.

For the elimination forest guided approach, we enforce a constraint so that the above size of extra buffer space ($\frac{2.5 \cdot BSIZE}{n} \cdot S_1$) is also sufficient. This constraint is that for any processor that executes both $Factor(k)$ and $Factor(k')$ where $k < k'$, $Factor(k')$ cannot start until $Factor(k)$ completes. In other words, $Factor()$ tasks are executed sequentially on each single processor column but they can be concurrent across all processor columns. As a result, our parallel algorithm is space-scalable for handling large matrices. Allocating more buffers can relax the above constraint and increase the degree of stage overlapping. However, our current experimental study does not show a substantial advantage by doing that and we plan to investigate this issue further in the future. Figure 8 shows our elimination forest guided approach based on the above strategy.

Example. Figure 9(a) and (b) are the factor-ahead and elimination forest guided schedules for the partitioned matrix in Figure 5(b) on a 2×2 processor grid. Notice that some of $Update2D()$ tasks such as $U(1, 2)$ are not listed because they do not exist due to the matrix sparsity. To simplify our illustration, we assume that each of $Factor()$, $ScaleSwap()$ and $Update2D()$ takes one unit time and communication cost is zero. In the factor-ahead schedule, $Factor(3)$ is executed immediately after $Update2D(1, 3)$ on the processor column 1. The basic approach would schedule $Factor(3)$ after $ScaleSwap(2)$. Letting $Factor()$ tasks complete as early as possible is important since many updating tasks depend on $Factor()$ tasks. In the elimination

```
Let (my_rno, my_cno) be the 2D coordinates of this processor;
Let m be the smallest column block number owned by this
    processor.
if m doesn't have any child supernode then
   Perform task Factor(m) for blocks this processor owns;
endif
for k = 1 to N − 1
   Perform ScaleSwap(k) for blocks this processor owns;
   Let m be the smallest column block number (m > k) this
      processor owns.
   Perform Update2D(k, m) for blocks this processor owns;
   if column block m is not factorized
      and all m's child supernodes have been factorized then
         Perform Factor(m) for blocks this processor owns;
   endif
   for j = m + 1 to N
      if my_cno = j mod p_c then
         Perform Update2D(k, j) for blocks this processor owns;
      endif
   endfor
endfor
```

FIG. 8. *Supernode elimination forest guided 2D approach.*

forest based schedule, $Factor(2)$ is executed in parallel with $Factor(1)$ because there is no dependence between them, represented by the forest in Figure 7. As a result, the length of this schedule is one unit shorter than the factor-ahead schedule.

PC1	PC2
F(1)	Idle
S(1)	S(1)
U(1,3)	F(2)
F(3)	U(1,4)
S(2)	S(2)
S(3)	U(2,4)
Idle	S(3)
Idle	U(3,4)
Idle	F(4)

(a) Factor-ahead Approach

PC1	PC2
F(1)	F(2)
S(1)	S(1)
U(1,3)	U(1,4)
F(3)	S(2)
S(2)	U(2,4)
S(3)	S(3)
Idle	U(3,4)
Idle	F(4)

(b) Elimination Forest Guided Approach

FIG. 9. *Task schedules for matrix in Figure 5(b). $F()$ stands for $Factor()$, $S()$ stands for $ScaleSwap()$, $U()$ stands for $Update2D()$ and PC stands for Processor Column.*

5. Fast supernodal GEMM kernel. We examine how the computation-dominating part of the LU algorithm can be efficiently implemented using the highest level of

BLAS possible. Computations in task $Update2D()$ involve the following supernode block multiplication: $A_{i,j} = A_{i,j} - L_{i,k} * U_{k,j}$ where $k < i$ and $k < j$. As we mentioned in the end of Section 3.2, submatrices like $A_{i,j}$, $L_{i,k}$ and $U_{k,j}$ are all stored in a compressed storage scheme with bit maps which identify their dense subcolumns or subrows. As a result, the BLAS-3 GEMM routine [7] may not be directly applicable to $A_{i,j} = A_{i,j} - L_{i,k} * U_{k,j}$ because subcolumns or subrows in those submatrices may not be consecutive and the target block $A_{i,j}$ may have a nonzero structure different from that of product $L_{i,k} * U_{k,j}$.

There could be several approaches to circumvent the above problem. One approach is to use a mixture of BLAS-1/2/3 routines. If $L_{i,k}$ and $A_{i,j}$ have the same row sparse structure, and $U_{k,j}$ and $A_{i,j}$ have the same column sparse structure, BLAS-3 GEMM can be directly used to modify $A_{i,j}$. If only one of the above two conditions holds, then the BLAS-2 routine GEMV can be employed. Otherwise only the BLAS-1 routine DOT can be used. In the worst case, the performance of this approach is close to the BLAS-1 performance. Another approach is to treat non-zero submatrices of A as dense during space allocation and submatrix computation, and hence BLAS-3 GEMM can be employed more often. But considering the average density of submatrices is only around 51% for our test matrices, this approach normally leads to an excessive amount of extra space and unnecessary arithmetic operations.

We propose the following approach called *Supernodal GEMM* to minimize unnecessary computation but retain high efficiency. The basic idea is described as follows. If the BLAS-3 GEMM is not directly applicable, we divide the operation into two steps. At the first step, we ignore the target nonzero structure of $A_{i,j}$ and directly use BLAS-3 GEMM to compute $L_{i,k} * U_{k,j}$. The result is stored in a temporary block. At the second step, we merge this temporary block with $A_{i,j}$ using subtraction. Figure 10 illustrates these two steps. Since the computation of the first step is more expensive than the second step, our code for multiplying supernodal submatrices can achieve performance comparable to BLAS-3 GEMM. A further optimization is to speed-up the second step since the result merging starts to play some role for the total time after the GEMM routine reduces the cost of the first step. Our strategy is that if the result block and $A_{i,j}$ have the same row sparse structure or the same column sparse structure, the BLAS-1 AXPY routine should be used to avoid scalar operations. And to increase the probability of structure consistency between the temporary result block and $A_{i,j}$, we treat some of L and U submatrices as dense during the space allocation stage if the percentage of nonzeros in such a submatrix compared to the entire block size exceeds a threshold. For Cray-T3E, our experiments show that threshold 85% is the best to reduce the result merging time with small space increase.

6. Experimental studies on Cray T3E. S^+ has been implemented on Cray T3E using its SHMEM communication library. Most of our experiments are conducted on a T3E machine at San Diego Supercomputing Center (SDSC). Each Cray-T3E processing element at SDSC has a clock rate of 300MHz, an 8Kbytes internal cache, 96Kbytes second level cache, and 128Mbytes memory. The peak bandwidth between nodes is reported as 500Mbytes/s and the peak round trip communication latency is about 0.5-2μs [28]. We have observed that when the block size is 25, double-precision GEMM achieves 388MFLOPS while double precision GEMV reaches 255MFLOPS. We have used a block size 25 in our experiments. We also obtained access to a Cray-T3E at the NERSC division of the Lawrence Berkeley Lab. Each node in this machine has a clock rate of 450MHz and 256Mbytes memory. We have done one set

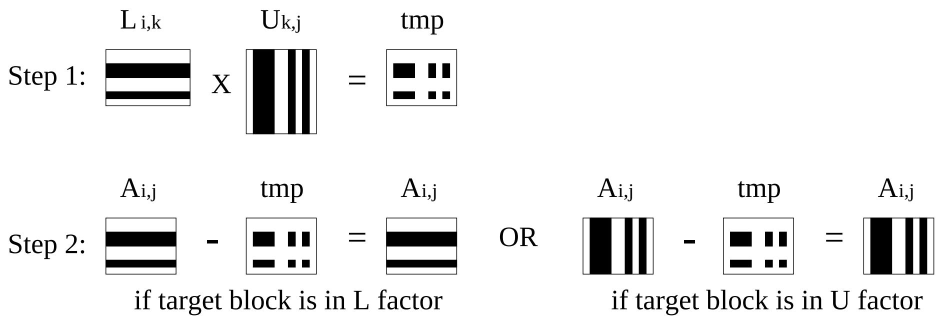

FIG. 10. *An illustration of Supernodal GEMM. Target block $A_{i,j}$ could be in the L part or U part.*

of experiments to show the performance improvement on this faster machine.

In this section, we report the overall sequential and parallel performance of S^+ without incorporating space optimization techniques, and measure the effectiveness of the optimization strategies proposed in Sections 3 and 4. In next section, we will study the memory requirement of S^+ with and without space optimization. Table 1 shows the statistics of the test matrices used in this section. Column 2 is the orders of the matrices and column 3 is the number of nonzeros before symbolic factorization. In column 4, 5 and 6 of this table, we have also listed the total number of nonzero entries divided by $|A|$ using three methods. Those nonzero entries including fill-ins are produced by dynamic factorization, static symbolic factorization, or Cholesky factorization of A^TA. The result shows that for these tested matrices, the total number of nonzeros predicted by static factorization is within 40% of what dynamic factorization produces. But the A^TA approach overestimates substantially more nonzeros, which indicates that the elimination tree of A^TA can introduce too many false dependency edges. All matrices are ordered using the minimum degree algorithm [3] on A^TA and the permutation algorithm for zero-free diagonal [8].

			$\frac{\text{factor entries}}{\lvert A\rvert}$			
Matrix	Order	$\lvert A\rvert$	Dynamic	Static	A^TA	Application domain
sherman5	3312	20793	12.03	15.70	20.42	Oil reservoir modeling
lnsp3937	3937	25407	17.87	27.33	36.76	Fluid flow modeling
lns3937	3937	25407	18.07	27.92	37.21	Fluid flow modeling
sherman3	5005	20033	22.13	31.20	39.24	Oil reservoir modeling
jpwh991	991	6027	23.55	34.02	42.57	Circuit physics
orsreg1	2205	14133	29.34	41.44	52.19	Oil reservoir simulation
saylr4	3564	22316	30.01	44.19	56.40	Oil reservoir modeling
goodwin	7320	324772	9.63	10.80	16.00	Fluid mechanics
e40r0100	17281	553562	14.76	17.32	26.48	Fluid dynamics
raefsky4	19779	1316789	20.36	28.06	35.68	Container modeling
inaccura	16146	1015156	9.79	12.21	16.47	Structure problem
af23560	23560	460598	30.39	44.39	57.40	Navier-Stokes Solver
fidap011	16614	1091362	23.36	24.55	31.21	Finite element modeling
vavasis3	41092	1683902	29.21	32.03	38.75	PDE

TABLE 1
Test matrices and their statistics.

[3]A Matlab program is used for minimum degree ordering.

In calculating the MFLOPS achieved by our parallel algorithm, *we do not include extra floating point operations introduced by static fill-in overestimation and supernode amalgamation.* The achieved MFLOPS is computed by using the following formula:

$$\text{Achieved MFLOPS} = \frac{\text{True operation count}}{\text{Elapsed time of our algorithm on the T3E}}.$$

The true operation count is obtained by running SuperLU without amalgamation. Amalgamation can be turned off in SuperLU by setting the relaxation parameter for amalgamation to 1 [6, 24].

6.1. Overall code performance. Table 2 lists the sequential performance of S^+, our previous design S^*, and SuperLU [4]. The result shows S^+ can actually be faster than SuperLU because of the use of new supernode partitioning and matrix multiplication strategies. The test matrices are selected from Table 1 that can be executed on a single T3E node. The performance improvement ratios from S^* to S^+ vary from 22% to 40%. Notice that time measurement in Table 2 excludes symbolic preprocessing time. However, symbolic factorization in our algorithms is very fast and takes only about 4.2% of numerical factorization time for the matrices in Table 2. And this ratio tends to decrease as the matrix size increases. This preprocessing cost is insignificant, especially when LU factorization is used in an iterative algorithm. In Table 2, we list the time of symbolic factorization for each matrix inside the parentheses behind the time of S^+.

Matrix	Sequential S^+		SuperLU		Sequential S^*		Time Ratio	
	Time	Mflops	Time	Mflops	Time	Mflops	$\frac{S^+}{SuperLU}$	$\frac{S^+}{S^*}$
sherman5	0.65 (0.04)	38.6	0.78	32.2	0.94	26.7	0.83	0.69
lnsp3937	1.48 (0.08)	22.9	1.73	19.5	2.00	16.9	0.86	0.74
lns3937	1.58 (0.09)	24.2	1.84	20.8	2.19	17.5	0.86	0.72
sherman3	1.56 (0.03)	36.2	1.68	33.6	2.03	27.8	0.93	0.77
jpwh991	0.52 (0.03)	31.8	0.56	29.5	0.69	23.9	0.93	0.75
orsreg1	1.60 (0.04)	35.0	1.53	36.6	2.04	27.4	1.05	0.78
saylr4	2.67 (0.07)	37.2	2.69	36.9	3.53	28.1	0.99	0.76
goodwin	10.26 (0.35)	65.2	-	-	17.0	39.3	-	0.60

TABLE 2

Sequential performance on a 300MHz Cray T3E node. Symbol "-" implies the data is not available due to insufficient memory or is not meaningful due to paging.

For parallel performance, we compare S^+ with a previous version [10] in Table 3 and the improvement ratio in terms of MFLOPS varies from 16% to 116%, in average more than 50%. Table 4 shows the absolute performance of S^+ on the Cray T3E machine with 450MHz CPU. The highest performance reached is 10.85GFLOPS, while for the same matrix, 8.25GFLOPS is reached on the T3E with 300MHz processors".

6.2. Effectiveness of the proposed optimization strategies. Elimination forest guided partitioning and amalgamation. Our new strategy for supernode partitioning with amalgamation clusters columns and rows simultaneously using structural containment information implied by an elimination forest. Our previous design S^* [10, 11] does not consider the bounding of nonzeros in the U part. We compare

[4]We did not compare with another well-optimized package UMFPACK [2] because SuperLU has been shown competitive to UMFPACK [4].

Matrix	P=8		P=16		P=32		P=64		P=128	
	S^*	S^+	S^*	S^+	S^*	S^+	S^*	S^+	S^*	S^+
goodwin	215.2	403.5	344.6	603.4	496.3	736.0	599.2	797.3	715.2	826.8
e40r0100	205.1	443.2	342.9	727.8	515.8	992.8	748.0	1204.8	930.8	1272.8
raefsky4	391.2	568.2	718.9	1072.5	1290.7	1930.3	2233.3	3398.1	3592.9	5133.6
inaccura	272.2	495.5	462.0	803.8	726.0	1203.6	1172.7	1627.6	1524.5	1921.7
af23560	285.4	432.1	492.9	753.2	784.3	1161.3	1123.2	1518.9	1512.7	1844.7
fidap011	489.3	811.2	878.1	1522.8	1524.3	2625.0	2504.4	4247.6	3828.5	6248.4
vavasis3	795.5	937.3	1485.5	1823.7	2593.5	3230.8	4406.3	5516.2	6726.6	8256.0

TABLE 3
MFLOPS performance of S^+ and S^* on the 300MHz Cray T3E.

Matrix	P=8		P=16		P=32		P=64		P=128	
	Time	Mflops	Time	Mflops	Time	Mflops	Time	Mflops	Time	Mflops
goodwin	1.21	552.6	0.82	815.4	0.69	969.0	0.68	983.2	0.67	997.9
e40r0100	4.06	609.4	2.50	989.7	1.87	1323.2	1.65	1499.6	1.59	1556.2
raefsky4	38.62	814.6	20.61	1526.3	11.54	2726.0	6.80	4626.2	4.55	6913.8
inaccura	6.56	697.2	4.12	1110.1	2.80	1633.4	2.23	2050.9	1.91	2394.6
af23560	10.57	602.1	6.17	1031.5	4.06	1567.5	3.47	1834.0	2.80	2272.9
fidap011	21.58	1149.5	11.71	2118.4	6.81	3642.7	4.42	5612.3	3.04	8159.9
vavasis3	62.68	1398.8	33.68	2603.2	19.26	4552.3	11.75	7461.9	8.08	10851.1

TABLE 4
Experimental results of S^+ on the 450MHz Cray T3E. All times are in seconds.

our new code S^+ with a modified version using the previous partitioning strategy. The performance improvement ratio by using the new strategy is listed in Figure 11 and an average of 20% improvement is obtained. The ratio for matrix "af23560" is not substantial because this matrix is very sparse and the partitioning/amalgamation strategy cannot produce large supernodes.

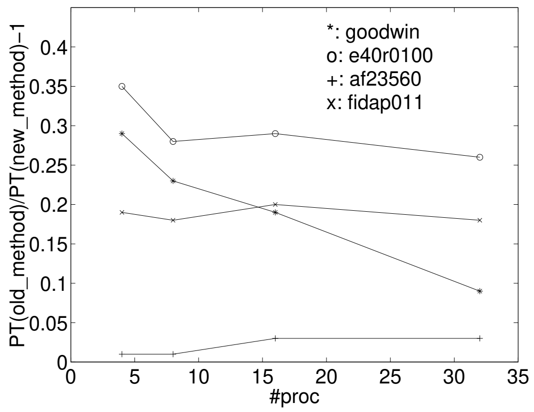

FIG. 11. *Performance improvement by using new supernode partitioning/amalgamation strategy.*

Effectiveness of supernodal GEMM. We assess the gain due to the introduction of our supernodal GEMM operation. We compare S^+ with a modified version using an approach which mixes BLAS-1/2/3 as described in Section 5. We do not compare with the approach that treats all nonzero blocks as dense since it introduces too much extra space and computation. The performance improvement ratio of our supernodal approach over the mixed approach is listed in Figure 12. The improvement is not substantial for matrix "e40r0100" and none for "goodwin". This is because they are relatively dense and the mixed approach has been employing BLAS-3 GEMM most of the time. For the other two matrices which are relatively sparse, the improvement ratio can be up to 10%.

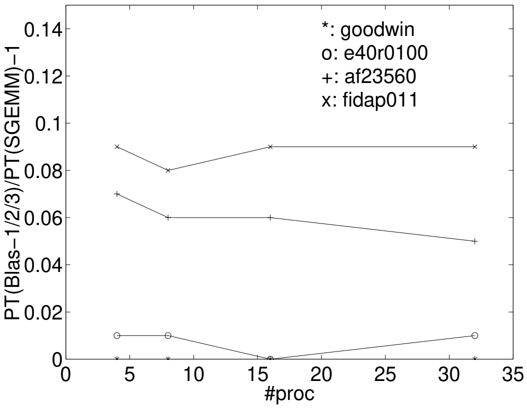

FIG. 12. *Performance improvement by using the supernodal GEMM.*

A comparison of the control strategies for exploiting 2D parallelism. In Table 5 we assess the performance improvement by using the elimination forest guided approach against the factor-ahead and basic approaches described in Section 4. Compared to the basic approach, the improvement ratios vary from 16% to 41% and the average is 28%. Compared to the factor-ahead approach, the average improvement ratio is 11% and the ratios tend to increase when the number of processors increases. This result is expected in the sense that the factor-ahead approach improves the degree of computation overlapping by scheduling factor tasks one step ahead while using elimination forests can exploit more parallelism.

Matrix	Improvement over Basic				Improvement over Factor-ahead			
	P=16	P=32	P=64	P=128	P=16	P=32	P=64	P=128
goodwin	41%	35%	19%	21%	8%	12%	10%	14%
e40r0100	38%	40%	30%	27%	15%	17%	12%	15%
raefsky4	21%	21%	34%	34%	7%	10%	11%	13%
inaccura	21%	28%	26%	27%	7%	13%	9%	13%
af23560	31%	37%	32%	30%	10%	15%	10%	13%
fidap011	24%	28%	36%	38%	8%	12%	11%	15%
vavasis3	17%	16%	31%	28%	3%	6%	8%	12%

TABLE 5
Performance improvement by using the elimination forest guided approach.

7. Space Optimization. For all matrices tested above static symbolic factorization provides fairly accurate prediction of nonzero patterns and only creates 10% to 50% more fill-ins compared to dynamic symbolic factorization used in SuperLU. However, for some matrices especially in circuit and device simulation, static symbolic factorization creates too many fill-ins. Table 6 shows characteristics of five matrices from circuit and device simulations. Static symbolic factorization does produce a large number of fill-ins for these matrices (up to 3 times higher than dynamic symbolic factorization using the same matrix ordering [5]). Our solution needs to provide a smooth adaptation in handling such cases.

			factor entries/$\|A\|$		
Matrix	Order	$\|A\|$	Dynamic	Static	$A^T A$
TIa	3432	25220	24.45	42.49	307.1
TId	6136	53329	27.53	61.41	614.2
TIb	18510	145149	91.84	278.34	1270.7
memplus	17758	99147	71.26	168.77	215.19
wang3	26064	177168	197.30	298.12	372.71

TABLE 6
Circuit and device simulation test matrices and their statistics.

For the above cases, we find that a significant number of predicted fill-ins 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 symbolic factorization excessively over-predicts fill-ins. Another strategy 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 a potential to save significantly more space since the early identification of zero blocks prevents their propagation in the update phase of numerical factorization.

Space requirements. We have conducted experiments [20] to study memory requirement by incorporating the above space optimization strategies into S^+ on a SUN Ultra-1 with 320MB memory. In the following study, we refer to the revised S^+ with space optimization as $SpaceS^+$. Table 7 lists the space requirement of S^+, SuperLU and $SpaceS^+$ for the matrices from Tables 1 and 6. Matrix vavasis3 is not listed because its space requirement is too high for all three algorithms on this machine.

The result in Table 7 shows that our space optimization strategies are effective. $SpaceS^+$ uses slightly less space compared to S^+ for matrices in Table 1 and 37% less space on average for matrices in Table 6 (68% less space for matrix TIb). Compared to SuperLU, our algorithm actually uses 3.9% less space on average while static symbolic factorization predicts 38% more nonzeros. That is because the U structure in SuperLU is less regular than that in S^+ and the indexing scheme in S^+ is simpler. Notice that the space cost in our evaluation includes symbolic factorization. This part of cost ranges from 1% to 7% of the total cost. We also list the ratio of $SpaceS^+$ processing time to S^+ and to SuperLU. Some entries are marked '-' instead of actual numbers because we observed paging on these matrices which may affect the accuracy of the result. In terms of average time cost, the new version is faster than SuperLU, which is consistent to the results in Section 6.1. It is also slightly faster than S^+ because the early elimination of zero blocks prevents their propagation and hence reduces unnecessary computation.

[5]Using a different matrix ordering (MMD on $A^T + A$), SuperLU generates fewer fill-ins on certain matrices. This paper focuses algorithm design when ordering is given and studies performance using one ordering method. An interesting future research topic is to study ordering methods that optimize static factorization.

Matrix	Space Requirement			Space Ratio		Time Ratio	
	S^+	SuperLU	$SpaceS^+$	$\frac{SpaceS^+}{SuperLU}$	$\frac{SpaceS^+}{S^+}$	$\frac{SpaceS^+}{SuperLU}$	$\frac{SpaceS^+}{S^+}$
sherman5	3.061	3.305	2.964	0.90	0.97	0.853	0.959
sherman3	5.713	5.412	5.536	1.02	0.97	1.023	0.944
orsreg1	5.077	4.555	4.730	1.04	0.93	0.920	0.823
saylr4	8.509	7.386	8.014	1.09	0.94	0.964	0.870
goodwin	29.192	35.555	28.995	0.82	0.99	0.657	0.993
e40r0100	79.086	93.214	78.568	0.84	0.99	-	-
raefsky4	303.617	272.947	285.920	1.05	0.94	0.707	0.921
af23560	170.166	147.307	162.839	1.11	0.96	0.869	0.984
fidap011	221.074	271.423	219.208	0.81	0.99	-	-
TIa	8.541	6.265	7.321	1.17	0.86	0.675	0.629
TId	29.647	18.741	19.655	1.05	0.66	0.366	0.366
memplus	138.218	75.194	68.467	0.91	0.50	-	-
TIb	341.418	221.285	107.711	0.49	0.32	-	-
wang3	430.817	-	347.505	-	0.81	-	-

TABLE 7

Space requirement in MBytes on a SUN Ultra-1 machine. Symbol "-" implies that the data is not available due to insufficient memory or paging which affects the measurement.

Matrix	vavasis3	TIa	TId	TIb	memplus	wang3
MFLOPS on 128 nodes	10004.0	739.9	1001.9	2515.7	6548.4	6261.0
MFLOPS on 8 nodes	1492.9	339.6	281.5	555.7	1439.4	757.8

TABLE 8

MFLOPS performance of $SpaceS^+$ on 450MHz Cray T3E.

Parallel Performance. Our experiments on Cray T3E show that the parallel time performance of *SpaceS*$^+$ is still competitive to S^+. Table 8 lists performance of *SpaceS*$^+$ on vavasis3 and circuit simulation matrices in 450MHz T3E nodes. *SpaceS*$^+$ can still achieve 10.00GFLOPS on matrix vavasis3, which is not much less than the highest 10.85GFLOPS achieved by S^+ on 128 450MHz T3E nodes. For circuit simulation matrices, *SpaceS*$^+$ delivers reasonable performance.

Table 9 is the time difference of S^+ with and without space optimization on 300Mhz T3E nodes. For the matrices with high fill-in overestimation ratios, we observe that S^+ with dynamic space management is better than S^+. It is about 109% faster on 8 processors and 18% faster on 128 processors. As for other matrices, on 8 processors *SpaceS*$^+$ is about 1.24% slower than S^+ while on 128 processors, it is 7% slower than S^+. On average, *SpaceS*$^+$ tends to become slower when the number of processors becomes larger. This is because the lazy space 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.

8. Concluding remarks. Our experiments show that the proper use of elimination forests allows for effective matrix partitioning and parallelism exploitation. Together with the supernodal matrix multiplication algorithm, our new design can improve the previous code substantially and set a new performance record. Our experiments also show that S^+ with dynamic space optimization can deliver high performance for large sparse matrices with reasonable memory cost. Static symbolic factorization may create too many fill-ins for some matrices, but our space optimiza-

Matrix	P=8	P=16	P=32	P=64	P=128
goodwin	-7.28%	-8.29%	-8.10%	-1.17%	-4.69%
e40r0100	-6.81%	-8.81%	-11.34%	-8.84%	-7.13%
raefsky4	3.41%	2.52%	-0.42%	-1.82%	-5.02%
af23560	-3.17%	-3.98%	-9.72%	-4.56%	-13.76%
vavasis3	7.65%	-1.79%	5.02%	-2.16%	-6.13%
TIa	13.16%	10.42%	2.63%	-2.94%	-6.06%
TId	35.15%	23.81%	9.28%	-2.50%	-9.59%
TIb	352.20%	270.26%	209.27%	133.69%	78.10%
memplus	136.43%	115.38%	87.09%	61.84%	35.24%
wang3	10.51%	7.57%	3.52%	1.53%	-5.17%

TABLE 9
Performance difference of S^+ and SpaceS^+ on 300MHz Cray T3E. A positive number indicates an improvement of SpaceS^+ over the original S^+, while a negative number indicates a slowdown.

tion techniques can effectively reduce memory requirements. Our comparison with SuperLU indicates that the sequential version of S^+ is highly optimized and can be faster than SuperLU. Our evaluation has focused on using a simple, but popular ordering strategy. Different matrix reordering methods can result in different numbers of fill-ins. More investigation is needed to address this issue in order to reduce overestimation ratios.

Performance of S^+ is sensitive to the underlying message-passing library performance. Our experiments use the SHMEM communication library on Cray T3E and recently we have implemented S^+ using MPI 1.1. The MPI based S^+ version is more portable, however the current version is about 30% slower than the SHMEM based version. This is because SHMEM uses direct remote memory access while MPI requires hand-shake between communication peers, which involves synchronization overhead. We expect that more careful optimization on this MPI version can lead to better performance and use of one-side communication available in the future MPI-2 release may also help boosting performance. The source code of this MPI-based S^+ version is available at http://www.cs.ucsb.edu/research/S+ and the HPC group in SUN Microsystems plans to include it in their next release of the S3L library used for SUN SMPs and clusters [22].

Acknowledgments. We would like to thank Bin Jiang and Steven Richman for their contribution in implementing S^+, Horst Simon for providing access to a Cray T3E at the National Energy Research Scientific Computing Center, Stefan Boeriu for supporting access to a Cray T3E at San Diego Supercomputing Center, Andrew Sherman and Vinod Gupta for providing circuit simulation matrices, Tim Davis, Apostolos Gerasoulis, Xiaoye Li, Esmond Ng, and Horst Simon for their help during our research, and the anonymous referees for their valuable comments.

Appendix A. Notations.

A	The sparse matrix to be factorized. Notice that elements of A change during factorization. In this paper proposed optimizations are applied to A after symbolic factorization.
$a_{i,j}$	The element in A with row index i and column index j.
$a_{i:j,s:t}$	The submatrix in A from row i to row j and from column s to t.
l_k	Column k in the low triangular part of A.
u_k	Row k in the upper triangular part of A.

$\hat{a}_{i,j}$	$\hat{a}_{i,j} \neq 0$ if and only if $a_{i,j}$ is nonzero after symbolic factorization.
$\hat{l}_k$	The index set of nonzero elements in l_k after symbolic factorization.
$\hat{u}_k$	The index set of nonzero elements in u_k after symbolic factorization.
$\lvert\hat{l}_k\rvert$	The cardinality of $\hat{l}_k$.
$\lvert\hat{u}_k\rvert$	The cardinality of $\hat{u}_k$.
$A_{i,j}$	The submatrix in the partitioned A with row block index i and column block index j.
$A_{i:j,s:t}$	The submatrices in the partitioned A from row block i to j and from column block s to t.
$L_{i,j}$	The submatrix with block index i and j in the lower triangular part.
$U_{i,j}$	The submatrix with block index i and j in the upper triangular part.
$R(i:j)$	Relaxed L/U supernode, which contains a diagonal block, an L supernode and a U supernode.

Appendix B. Proof of Theorems.

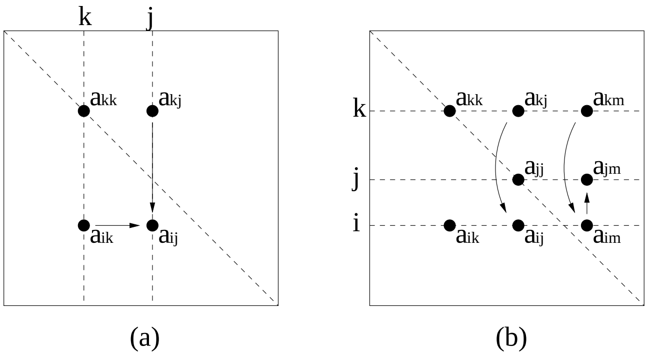

FIG. 13. *An illustration for the proof of Theorem 3.2.*

B.1. Theorem 1. *Proof.* To prove the theorem holds when vertex j is an *ancestor* of vertex k, we need only to show that it holds if vertex j is the *parent* of vertex k, because of the transitivity of "$\subseteq$".

If vertex j is the parent of vertex k in this elimination forest, $\hat{a}_{k,j} \neq 0$. Let $a^t_{i,j}$ denote the symbolic value of $a_{i,j}$ after step t of symbolic factorization. Since $a_{k,j}$ is not changed after step k of symbolic factorization, $a^k_{k,j} \neq 0$.

We first examine the L part as illustrated in Figure 13(a). For any $i > k$ and $i \in \hat{l}_k$, i.e., $\hat{a}_{i,k} \neq 0$, we have $a^k_{i,k} \neq 0$. Because $a^k_{i,k}$ and $a^k_{k,j}$ are used to update $a^k_{i,j}$, it holds that $i \in \hat{l}_j$. Therefore, $\{r \mid r \in \hat{l}_k \wedge j \leq r \leq n\} \subseteq \hat{l}_j$.

Next we examine the U part as illustrated in Figure 13(b). Since l_k must contain at least one nonzero off-diagonal element before step k of symbolic factorization, we assume it is $a^{k-1}_{i,k}$. Because $a_{k,j}$ is the first off-diagonal nonzero in $\hat{u}_k$, and $\hat{a}_{k,i} \neq 0$, we know $i \geq j$. For any $m > j$ and $m \in \hat{u}_k$, we prove $m \in \hat{u}_j$ as follows. Since $\hat{a}_{k,m} \neq 0$ and $a^k_{i,k} \neq 0$, it follows that $a^k_{i,j} \neq 0$ and $a^k_{i,m} \neq 0$. Therefore, $a^j_{i,j} \neq 0$. As a result, $a^k_{i,m} \neq 0$ and $a^j_{j,m} \neq 0$. And we conclude $\{c \mid c \in \hat{u}_k \wedge j \leq c \leq n\} \subseteq \hat{u}_j$. ☐

B.2. Theorem 2. *Proof.* If l_k directly updates l_j in LU factorization, vertex k must have a parent in the forest. Let

$$T = \{t \mid t \leq j \text{ and } t \text{ is an ancestor of } k \text{ in the elimination forest}\}$$

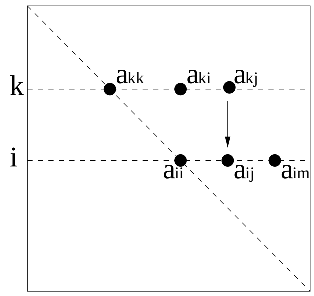

FIG. 14. *An illustration for the proof of Theorem 3.4.*

Since k's parent $\leq j$, set T is not empty. Let i be the largest element in T. We show $i = j$ by contradiction as illustrated in Figure 14. Assume $i < j$. Following Theorem 3.2, $\{c \mid c \in \hat{u}_k \wedge i \leq c \leq n\} \subseteq \hat{u}_i$. Since $\hat{a}_{k,j} \neq 0$, we know $\hat{a}_{i,j} \neq 0$. Let m be i's parent. Since i is the largest element in $T$and $m > i$, we know $m \notin T$. Thus, it holds that $m > j$. However, $a_{i,m}$ should be the first off-diagonal nonzero in $\hat{u}_i$, this is a contradiction since $\hat{a}_{i,j} \neq 0$. Thus vertex j is an ancestor of vertex k in the elimination forest.

If l_k indirectly updates l_j, there must be a sequence $s_1, s_2, \cdots, s_p$ such that: $s_1 = k$, $s_p = j$ and l_{s_q} directly updates $l_{s_{q+1}}$ for each $1 \leq q \leq p-1$. That is, vertex s_{q+1} is an ancestor of vertex s_q for each $1 \leq q \leq p-1$. Thus, we conclude that vertex j is an ancestor of vertex k.

Conversely, if vertex j is an ancestor of vertex k in the elimination forest, there must be a sequence $s_1, s_2, \cdots, s_p$ such that: $s_1 = k$, $s_p = j$ and vertex s_{q+1} is the parent of vertex s_q for each q where $1 \leq q \leq p-1$. Then for each $1 \leq q \leq p-1$, l_{s_q} directly updates $l_{s_{q+1}}$ since $|\hat{l}_{s_q}| \neq 1$ and $\hat{a}_{s_q, s_{q+1}} \neq 0$. Thus, we conclude that l_k directly or indirectly updates l_j during numerical factorization. ☐

B.3. Theorem 3. *Proof.* The "if" part is an immediate result of Definition 4.1. Now we prove the "only if" part. If $R(i_1 : i_2)$ is the parent of $R(j_1 : j_2)$ in the supernodal elimination forest, there exists vertex $i \in \{i_1, i_1 + 1, \cdots, i_2\}$ and vertex $j \in \{j_1, j_1 + 1, \cdots, j_2\}$ such that i is j's parent in the corresponding nodal elimination forest. Below we prove by contradiction that such a vertex j is unique and it must be j_2.

Suppose j is not j_2, i.e., $j_1 \leq j < j_2$. Since the diagonal block of $R(j_1 : j_2)$ is considered to be dense (including symbolic fill-ins after amalgamation), for every $u \in \{j_1, j_1 + 1, \cdots, j_2 - 1\}$, u's parent is $u + 1$ in the nodal elimination forest. Thus j's parent should be one in $\{j_1 + 1, \cdots, j_2\}$; however, we also know that j's parent is i in the nodal elimination forest and $j_2 < i$. That is a contradiction. ☐

B.4. Theorem 4. *Proof.* If the L part of supernode $R(j_1 : j_2)$ directly or indirectly updates L supernode $R(i_1 : i_2)$, there exists an l_j ($j \in \{j_1, j_1 + 1, \cdots, j_2\}$) which directly or indirectly updates column l_i ($i \in \{i_1, i_1 + 1, \cdots, i_2\}$). Because of Theorem 3.4, i is an ancestor of j. According to Definition 4.1, $R(i_1 : i_2)$ is an ancestor of supernode $R(j_1 : j_2)$.

On the other hand, if $R(i_1 : i_2)$ is an ancestor of supernode $R(j_1 : j_2)$, for each child-parent pair in the path from $R(j_1 : j_2)$ to $R(i_1 : i_2)$, we can apply both Theorem 4.2 and Theorem 3.4. Then, it is easy to show that the L part of each child supernode in this path directly or indirectly updates the L part of its parent supernode. Thus L part of supernode $R(j_1 : j_2)$ directly or indirectly updates L part of supernode $R(i_1 : i_2)$. $\square$

REFERENCES

[1] C. Ashcraft, R. Grimes, J. Lewis, B. Peyton, and H. Simon, *Progress in Sparse Matrix Methods for Large Sparse Linear Systems on Vector Supercomputers*, International Journal of Supercomputer Applications, 1 (1987), pp. 10–30.

[2] T. Davis and I. S. Duff, *An Unsymmetric-pattern Multifrontal Method for Sparse LU factorization*, SIAM Matrix Analysis & Applications, (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. S. Li, and J. W. H. Liu, *A Supernodal Approach to Sparse Partial Pivoting*, SIAM J. Matrix Anal. Appl., 20 (1999), pp. 720–755.

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

[6] J. W. Demmel, J. R. Gilbert, and X. S. Li, *SuperLU Users' Guide*, 1997.

[7] J. Dongarra, J. D. Croz, S. Hammarling, and R. Hanson, *An Extended Set of Basic Linear Algebra Subroutines*, ACM Trans. on Mathematical Software, 14 (1988), pp. 18–32.

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

[9] 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, (1997).

[10] ———, *Efficient Sparse LU Factorization with Partial Pivoting on Distributed Memory Architectures*, IEEE Transactions on Parallel and Distributed Systems, 9 (1998), pp. 109–125.

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

[12] ———, *Space and Time Efficient Execution of Parallel Irregular Computations*, in Proceedings of ACM Symposium on Principles & Practice of Parallel Programming, Las Vegas, June 1997.

[13] K. Gallivan, B. Marsolf, and H. Wijshoff, *The Parallel Solution of Nonsymmetric Sparse Linear Systems Using H* Reordering and an Associated Factorization*, in Proc. of ACM International Conference on Supercomputing, Manchester, July 1994, pp. 419–430.

[14] A. George and E. Ng, *Symbolic Factorization for Sparse Gaussian Elimination with Partial Pivoting*, SIAM, 8 (1987), pp. 877–898.

[15] ———, *Parallel Sparse Gaussian Elimination with Partial Pivoting*, Annals of Operations Research, 22 (1990), pp. 219–240.

[16] J. R. Gilbert and E. Ng, *Predicting structure in nonsymmetric sparse matrix factorizations*, Graph Theory and Sparse Matrix Computation (Edited by Alan George and John R. Gilbert and Joseph W. H. Liu), Springer-Verlag, 1993.

[17] G. Golub and J. M. Ortega, *Scientific Computing: An Introduction with Parallel Computing Compilers*, Academic Press, Boston, 1993.

[18] A. Gupta, G. Karypis, and V. Kumar, *Highly Scalable Parallel Algorithms for Sparse Matrix Factorization*, IEEE Transactions on Parallel and Distributed Systems, 8 (1995).

[19] S. Hadfield and T. Davis, *A Parallel Unsymmetric-pattern Multifrontal Method*, Tech. Report TR-94-028, Computer and Information Sciences Department, University of Florida, Aug. 1994.

[20] B. Jiang, S. Richman, K. Shen, and T. Yang, *Efficient Sparse LU Factorization with Lazy Space Allocation*, in Proceedings of the Ninth SIAM Conference on Parallel Processing for Scientific Computing, San Antonio, Texas, Mar. 1999.

[21] 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, Aug. 1997.

[22] G. Kechriotis, *Personal Communication*, 1999.

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

[24] ———, *Personal Communication*, 1998.

[25] J. W. H. Liu, *The Role of Elimination Trees in Sparse Factorization*, SIAM Journal on Matrix Analysis and Applications, 11 (1990), pp. 134–172.

[26] E. Rothberg, *Exploiting the Memory Hierarchy in Sequential and Parallel Sparse Cholesky Factorization*, PhD thesis, Dept. of Computer Science, Stanford, Dec. 1992.

[27] E. Rothberg and R. Schreiber, *Improved Load Distribution in Parallel Sparse Cholesky Factorization*, in Proc. of Supercomputing'94, Washington D.C., Nov. 1994, pp. 783–792.

[28] S. L. Scott and G. M. Thorson, *The Cray T3E Network: Adaptive Routing in a High Performance 3D Torus*, in Proceedings of HOT Interconnects IV, Stanford University, Aug. 1996.

[29] 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, Puerto Vallarta, Mexico, June 1998, pp. 5–15. Available on the World Wide Web at *http://www.cs.ucsb.edu/research/S+/*.