Elimination Forest Guided 2D Sparse LU Factorization *

Kai Shen
Dept. of Computer Science
University of California
Santa Barbara, CA 93106
kshen@cs.ucsb.edu

Xiangmin Jiao
Dept. of Computer Science
University of Illinois
Urbana-Champaign, IL 61801
jiao@cs.uiuc.edu

Tao Yang
Dept. of Computer Science
University of California
Santa Barbara, CA 93106
tyang@cs.ucsb.edu

Abstract

Sparse LU factorization with partial pivoting is important for many scientific applications and delivering high performance for this problem is difficult on distributed memory machines. Our previous work has developed an approach called S^* that incorporates static symbolic factorization, supernode partitioning and graph scheduling. This paper studies the properties of elimination forests and uses them to guide supernode partitioning/amalgamation and execution scheduling. The new design with 2D mapping effectively identifies dense structures without introducing too many zeros in the BLAS computation and exploits asynchronous parallelism with low buffer space cost. The implementation of this code, called S^+, uses supernodal matrix multiplication which retains the BLAS-3 level efficiency and avoids unnecessary arithmetic operations. The experiments show that S^+ improves our previous code substantially and can achieve up to 11.04GFLOPS on 128 Cray T3E 450MHz nodes, which is the highest performance reported in the literature.

1 Introduction

Solution of sparse linear systems is a computational bottleneck in many problems. If a matrix is symmetric and positive definite, Cholesky factorization can be used, for which fast parallel algorithms have been developed [15, 19, 20]. When pivoting is required to maintain numerical stability

*A short version of this paper will appear in the 10th annual ACM Symposium on Parallel Algorithms and Architectures for non-symmetric linear systems [2, 14], it is very hard to produce high performance for this problem because partial pivoting operations dynamically change computation and communication patterns during the elimination process, and cause severe caching miss and load imbalance on modern computers with memory hierarchies.

The previous work has addressed parallelization using shared memory platforms or restricted pivoting [3, 12, 13, 16]. Most notably, the recent shared memory implementation of SuperLU [3, 4, 18] has achieved up to 2.58GFLOPS on 8 Cray C90 nodes. For distributed memory machines, in [10] we proposed a novel approach called S^* that integrates three key strategies together in parallelizing this algorithm: 1) adopt a static symbolic factorization scheme [13] to eliminate the data structure variation caused by dynamic pivoting; 2) identify data regularity from the sparse structure obtained by the symbolic factorization so that efficient dense operations can be used to perform most of the computation; 3) make use of graph scheduling techniques and efficient run-time support called RAPID [9, 11] to exploit irregular parallelism. The preliminary experiments are encouraging and good performance results are obtained with 1D data mapping for a set of nonsymmetric benchmark matrices. We have achieved up to 1.35GFLOPS with RAPID code on 64 Cray T3E 300MHz nodes.

Our previous design uses task graphs for fast code prototyping. Elimination trees or forests are used extensively in sparse Cholesky because they have more compact representation of parallelism and can be used for both matrix partitioning and parallelism scheduling. It is difficult to handle sparse LU similarly for a general matrix A because A can be nonsymmetric and may require partial pivoting. The classical approach for LU normally uses elimination trees of A^TA, which normally produce too much false computational dependency. Thus our primary goal is to study the definition and properties of elimination trees/forests to guide matrix partitioning and parallelism control in LU.

Our second goal is to incorporate 2D block-based mapping in our framework. In the literature 2D mapping has been shown more scalable than 1D for dense LU and sparse Cholesky [1, 20, 21]. However there are difficulties to apply the 2D block-oriented mapping to the case of sparse LU factorization even the static structure is predicted in advance. First, pivoting operations and row interchanges require frequent and well-synchronized inter-processor communication when each column is distributed to multiprocessors. Second, exploiting irregular parallelism to a maximum degree may need a substantial amount of extra buffer space.

In [7], we reported a preliminary version of the 2D code with a simple parallelism scheduling mechanism. Recently with a modified control mechanism called factor-ahead, S^* has achieved up to 6.87GFLOPS on 128 Cray T3E 300MHz nodes [8]. In this paper, we will briefly explain this control mechanism, and will further report several new performance-improving strategies based on elimination forests. Those strategies include supernode partitioning and amalgamation using the properties of elimination forests, efficient supernode-level matrix multiplication, and parallelism exploitation using elimination forests. Our new design, called S^+, can improve our previous code in an average of more than 50% in terms of execution time. In particular we can achieve up to 8.44GFLOPS on 128 T3E 300MHz nodes and 11.04GFLOPS on 128 T3E 450MHz nodes. This is the highest performance ever achieved for this problem. Notice that extra arithmetic operations introduced by static factorization are *excluded* in computing GFLOPS rates of our algorithms.

The rest of this extended abstract is organized as follows. Section 2 gives the background knowledge of the sparse LU. Section 3 presents a modified definition and properties of elimination forests for sparse LU, gives the strategies of supernode partitioning and amalgamation using those properties. Section 4 describes strategies for 2D asynchronous parallelism exploitation. Section 5 discusses a fast matrix multiplication algorithm suitable for submatrices obtained by supernode partitioning strategies. Section 6 presents the experimental results on Cray T3E. Section 7 concludes the paper.

2 Background

The purpose of LU factorization is to find two matrices L and U for a nonsymmetric sparse matrix A 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. In this section, we briefly discuss related techniques used in our algorithm.

Static symbolic factorization. Static symbolic factorization is proposed in [13] 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. The symbolic factorization for an $n \times n$ matrix can be 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 $k - 1$ columns."

Using an efficient implementation of the symbolic factorization algorithm [17], 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 [3, 18], 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 [8, 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 and future work is needed to study re-ordering strategies to reduce over-estimation ratios.

L/U **supernode partitioning.** 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 [18], a nonsymmetric supernode is defined 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 [5] 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 [5]. Figure 1 illustrates an example of a partitioned sparse matrix and the black areas depict dense submatrices, subrows and subcolumns.

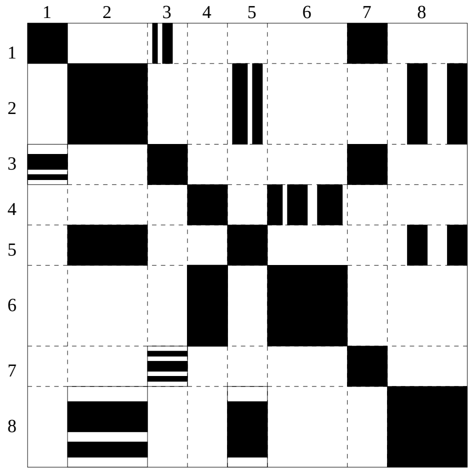

Figure 1: Example of a partitioned sparse matrix.

Data mapping. 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 row block index i and column block index j. For block-oriented matrix computation, 1D column block cyclic mapping and 2D block cyclic mapping are commonly used. In 1D column block cyclic mapping, the j-th column block of A is assigned to the same processor $P_{j \ mod \ p}$, where p is the number of the processors. In the 2D 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 [7, 17]. However 2D mapping introduces more overhead for pivoting and row swapping.

Program partitioning. Each column block k is associated with two types of tasks: $Factor(k)$ and $Update(k, j)$ for $1 \leq k < j \leq N$. 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" [2]. 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.

The 1D RAPID code. We have implemented a parallel method with 1D data mapping using the RAPID runtime system [9, 10]. This code uses a DAG to model irregular parallelism and RAPID to schedule the tasks. Then RAPID will execute the scheduled DAG on a distributed

```
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
```

Figure 2: Partitioned sparse LU factorization with partial pivoting.

memory machine using a low-overhead communication scheme. Using DAGs to model irregular LU parallelism is good in helping us understand the parallelism in sparse LU and develop the first prototype of high performance message-passing LU code. In [8, 17], we show that 1D RAPID code based on graph scheduling can actually outperform 2D codes with simpler scheduling methods when sufficient space is available. But 2D mapping exposes more parallelism, which makes 2D codes more scalable and easier to achieve load balance. Also the RAPID implementation in [10] uses extra memory space for supporting general irregular computations. Thus in designing 2D codes, we paid special attention to the usage of buffer space so that 2D codes are able to factorize large matrices under memory constraints.

3 Elimination forests and nonsymmetric supernode partitioning

In this section, we extend the previous work on elimination forests [1, 13] and identify the properties of elimination forests to design more robust strategies for supernode partitioning and detect when pivoting for different columns can be conducted concurrently. As a result, both sequential and parallel codes can be improved.

3.1 The definition of 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 [6]. We will use the following notations in the rest of this section. We will 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 1 *An* **LU Elimination forest** *for an $n \times n$ matrix A has n nodes 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 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. Figure 3 illustrates a sparse matrix after symbolic factorization and its elimination forest.

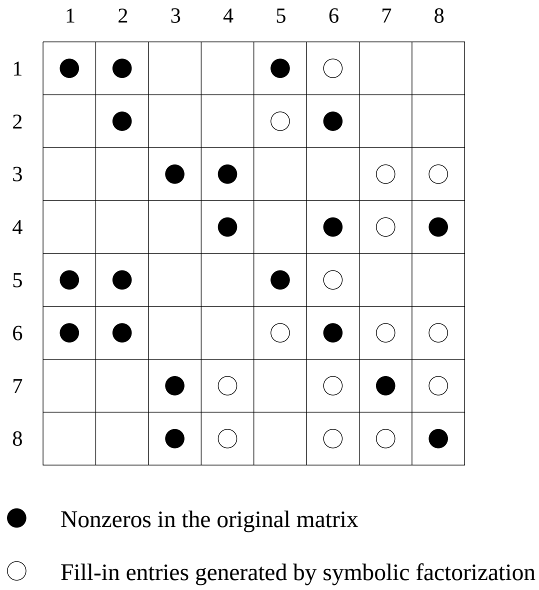

Figure 3: A sparse matrix and its elimination forest.

Theorem 1 below demonstrates the structural properties of an elimination forest.

Theorem 1 *If vertex j is an ancestor of vertex k in an elimination forest, then $L_k - \{k, k+1, \cdots, j-1\} \subseteq L_j$ and $U_k - \{k, k+1, \cdots, j-1\} \subseteq U_j$.*

Proof: First we prove the theorem when vertex j is the **parent** of vertex k in this elimination forest. The proof contains the following two parts:

- **Part (a)** (Illustrated in Figure 4(a))
  Let a_{ij}^t be the value of a_{ij} after step t of symbolic factorization. Since j is the parent of k, $a_{kj} \neq 0$. Also considering that the value of a_{kj} is not changed after step k of symbolic factorization, we know that $a_{kj}^k \neq 0$. Then for any $i > k$ that satisfies $a_{ik} \neq 0$, $a_{ik}^k \neq 0$ and $a_{ij}^k \neq 0$ following the nature of LU factorization. Therefore, $L_k - \{k, k+1, \cdots, j-1\} = L_k^k - \{k, k+1, \cdots, j-1\} \subseteq L_j^k \subseteq L_j$.

- **Part (b)** (Illustrated in Figure 4(b))
  Since j is the parent of k in the elimination forest, a_{kj}^k (or a_{kj}) is the first off-diagonal nonzero

in U_k and L_k must contain at least one off-diagonal element before step k of symbolic factorization, say $a_{ik}^{k-1} \neq 0$. We know $i \geq j$, otherwise $a_{ki} \neq 0$ contradicts the fact that "a_{kj} is the first off-diagonal nonzero in U_k". For any $m > j$ that satisfies $a_{km} \neq 0$, we prove $a_{jm} \neq 0$ by the following two steps:

1. Since $a_{km} = a_{km}^k$, $a_{km}^k \neq 0$. Also because $a_{ik}^k \neq 0$, row i is a candidate pivoting row at step k of LU factorization. Therefore, $a_{ij}^k \neq 0$ and $a_{im}^k \neq 0$.
2. Since $a_{ij}^k \neq 0$, row i is a candidate pivoting row at step j of LU factorization. Therefore, $a_{jm}^j \neq 0$ follows $a_{im}^k \neq 0$.

Then we conclude $U_k - \{k, k+1, \cdots, j-1\} \subseteq U_j$.

Thus Theorem 1 holds on the condition that vertex j is the **parent** of vertex k. According to the transitivity of "$\subseteq$", it is easy to conclude that Theorem 1 also holds under the condition that vertex j is an **ancestor** of vertex k in elimination forest. ■

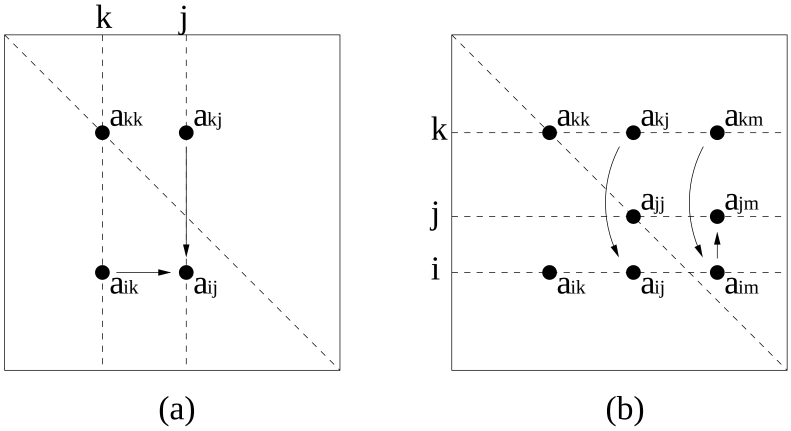

Figure 4: Illustration for the proof of Theorem 1.

Lemma 1 and Theorem 2 below identify the dependency information in the elimination forest.

Definition 2 *Let $j > k$, L_k* **directly updates** *L_j if task $Update(k, j)$ is performed in LU factorization, i.e. $a_{kj}^k \neq 0$ and $|L_k| > 1$. 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 $1 \leq q \leq p-1$.*

Lemma 1 *If L_k is used to directly update L_j in LU factorization, then vertex j is an ancestor of vertex k in the elimination forest.*

Proof: According to the nature of LU factorization, we know $k < j$, $a_{kj} \neq 0$ and $|L_k| > 1$. Let $T = \{t | t \leq j \text{ } and \text{ } t \text{ } is \text{ } an \text{ } ancestor \text{ } of \text{ } k \text{ } in \text{ } the \text{ } elimination \text{ } forest\}$. Set T is not empty because k's parent is in T. Let i be the largest element in T. As illustrated in Figure 5, we show $i = j$ by contradiction as follows. If $i \neq j$, then $i < j$ since every element in T is less than or equal to j. Following Theorem 1, $U_k - \{k, k+1, \cdots, i-1\} \subseteq U_i$ because i is an ancestor of k. Since $a_{kj} \neq 0$, $a_{ij} \neq 0$. Let m be i's parent, then $m > j$, otherwise $m \in T$ which contradicts the definition of i. But a_{im} should be the first off-diagonal nonzero in U_i, this is a contradiction since $a_{ij} \neq 0$. ■

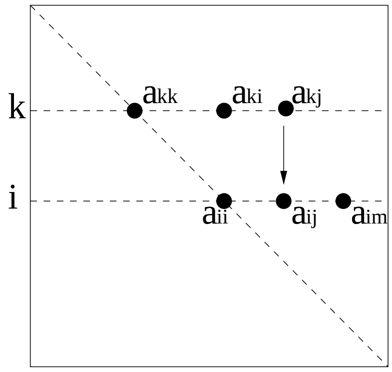

Figure 5: Illustration for the proof of Lemma 1.

Theorem 2 *Let $k < j$, L_k is used to directly or indirectly update L_j in LU factorization if and only if vertex j is an ancestor of vertex k.*

Proof: The proof includes the following two parts:

- If L_k directly or indirectly updates L_j in LU factorization, 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 $1 \leq q \leq p-1$. According to Lemma 1, we know 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.
- 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 $1 \leq q \leq p-1$. According to the nature of LU factorization, we know for each $1 \leq q \leq p-1$, L_{s_q} directly updates $L_{s_{q+1}}$ since $|L_{s_q}| \neq 1$ and $a_{s_q s_{q+1}} \neq 0$. Thus, we conclude that L_k directly or indirectly updates L_j in LU factorization.

■

Theorem 1 captures the structural containment between two columns in L and two rows in U, which will be used for designing supernode partitioning with amalgamation in the next subsection. Theorem 2 indicates dependency information in the numerical elimination, which can guide our parallel scheduling of asynchronous parallelism.

George and Ng proposed a definition of elimination forests in [13] to control row-wise elimination. The difference between their definition and the above definition is that we impose the condition $|L_k| > 1$. In practice, we find that the tested matrices can have up to 50% of columns with zero lower-diagonal elements. Imposing this condition avoids some unnecessary dependence among vertices and it is also required for proving Theorems 1 and 2.

Figure 6 illustrates the difference among three definitions of elimination tree/forests using a very simple example. Figure 6(a) shows a sparse matrix A and no fill-in is created by symbolic factorization. Figure 6(b) displays the elimination tree of $A^T A$. Figure 6(c) illustrates the elimination forest under George and Ng's definition and Figure 6(d) shows the elimination forest under our definition.

It can be seen that the elimination forest under our definition identifies more parallelism. Another observation is that Theorem 1 only holds under our definition of elimination forests.

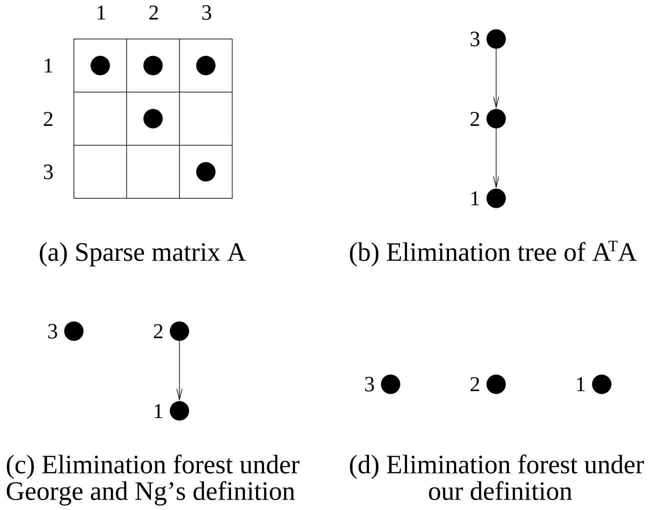

Figure 6: A sparse matrix and its elimination tree/forests under three different definitions.

3.2 2D L/U supernode partitioning and amalgamation

Given a nonsymmetric matrix A after symbolic factorization, in [10] we have described a 2D L/U supernode partitioning in which two stage partitioning is applied. Stage 1: A group of consecutive columns that have the same structure in the L factor is considered as one supernode column block. Then the L factor is sliced as a set of consecutive column blocks. Stage 2: After an L supernode partition has been obtained, the same partition is applied to the rows of the matrix to further break each supernode column block 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 1 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 1 *If for each $i \in \{s+1, s+2, \cdots, t\}$, vertex i is the parent of vertex $i-1$ and $|L_i| = |L_{i-1}| - 1$, then 1) the 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 can be 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 do not need to check any constraint on U because as long as a child-parent pair $(i, i-1)$ satisfies $|L_i| = |L_{i-1}| - 1$, we can show that $|U_i| = |U_{i-1}| - 1$ based on Theorem 1 in [10] and hence the structures of U_i and U_{i-1} are identical. Figure 7(a) illustrates supernode partitioning of the sparse matrix in Figure 3. There are 6 L/U supernodes and from the L partitioning point of view, columns from 1 to 5 are not grouped but columns 6, 7 and 8 are clustered together.

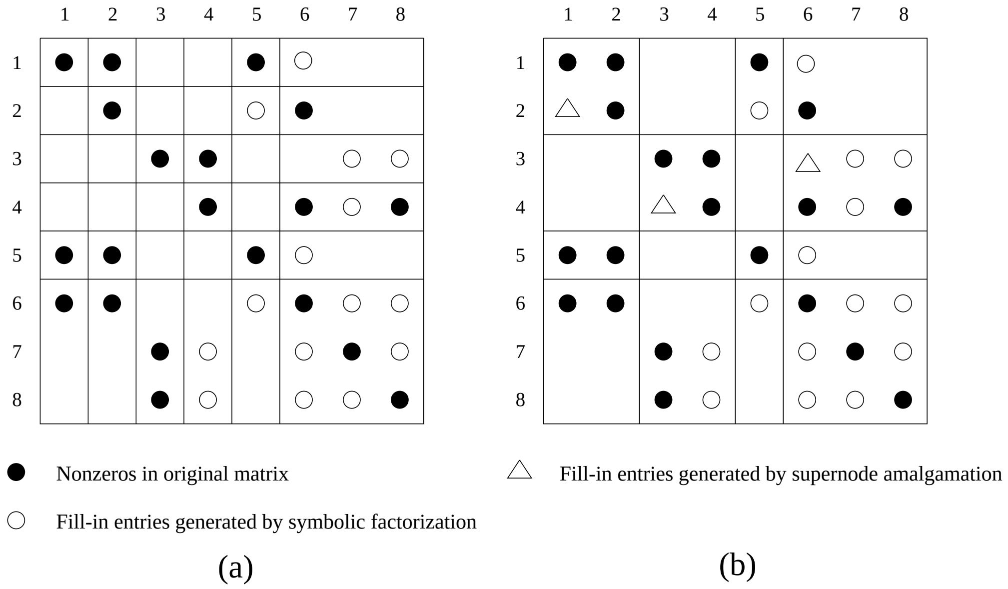

Figure 7: (a) Supernode partitioning for the matrix in Figure 3; (b) The result of supernode amalgamation.

For most of the tested sparse 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 dense structures of supernodes. In this way, caching performance can be improved and interprocessor communication overhead may be reduced. For sparse Cholesky (e.g. [19]), 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, such a permutation may alter the symbolic factorization result. In our previous approach [10], we simply compare the consecutive columns of the L factor, 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 L supernode. However, the result of partitioning may lead to too many extra zero entries in the dense structure of U supernode. Using the elimination forest properties, we can remedy this problem by partitioning L and U factors simultaneously as follows.

We call our supernodes after amalgamation as *relaxed L/U supernodes* and each of them includes elements from both the L part and the U part.

Definition 3 *A relaxed L/U supernode R(s:t) contains three parts: the diagonal block* $a_{s:t,s:t}$*, the L supernode part* $a_{s+1:n,s:t}$ *and the U supernode part* $a_{s:t,t+1:n}$.

The following corollary, which is also a straightforward result of Theorem 1, can be used to bound the nonzero structure of a relaxed L/U supernode.

Corollary 2 *If for each* i *where* $s+1 \le i \le t$*, vertex* i *is the parent of vertex* $i-1$ *in an elimination forest, then the nonzero structure of each column in* $a_{s+1:n,\ s:t}$ *is a subset of the structure in* L_t*, and the nonzero structure of each row in* $a_{s:t,\ t+1:n}$ *is a subset of the structure in* U_t.

Using Corollary 2, in $R(s:t)$ the ratio of extra fill-ins introduced by amalgamation compared with the actual nonzeros can be computed as:

$$z = \frac{(t-s+1)^2 + (t-s+1) \times (nz(L_t) + nz(U_t) - 2)}{nz(R(s:t))} - 1$$

where $nz()$ gives the number of nonzero elements in the corresponding structure including fill-ins created by symbolic factorization. Also notice that both L_t and U_t include the diagonal element.

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 $s+1 \le i \le t$, 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. $t-s+1 \le$ the pre-defined upper bound for supernode sizes.

Our experiments show that the above strategy is very effective and the complexity of the partitioning algorithm with amalgamation is $O(n)$, which is very low and is made possible by Corollary 2. Our experiments show that 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 T3E. Thus all the experiments in Section 6 are completed with $z \le 30\%$ and $t-s+1 \le 25$.

Figure 7(b) illustrates the result of supernode amalgamation for the sparse matrix in Figure 3. Condition $z \le 30\%$ is applied during the amalgamation. There are four relaxed L/U supernodes: $R(1:2)$, $R(3:4)$, $R(5:5)$, and $R(6:8)$.

In the rest of this paper, we will call relaxed L/U supernodes simply as supernodes, and the supernode size of $R(s:t)$ is $t-s+1$.

4 2D asynchronous parallelism exploitation

In this section, we present scheduling control 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. Symbol $A_{i,j}$ is used to denote the submatrix in row block i and column block j and $A_{i:j,s:t}$ denotes the submatrices from row block i to j and column block s to t. 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. One is $Factor(k)$, which is to factorize all the columns in the k-th 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 j-th column block, and modify the j-th column block using the k-th column block, where $k < j$ and $U_{kj} \neq 0$. The 2D data mapping enables parallelization of a single $Factor(k)$ or $Update(k, j)$ task on p_r processors because each of them is executed by column $k \bmod p_c$ of the processor grid. 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 a global synchronization between those processors 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 $A_{k:N,\ k+1:N}$; $Update2D(k, j)$ 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 [7] is a basic approach in which computation flow is controlled by the 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 $Facor(k+1)$ to start $Update2D(k+1, *)$. It is not necessary that $Facor(k+1)$ has to wait the completion of all tasks $Update2D(k, *)$. This idea has been used in the dense LU algorithm [14] and we extend it for asynchronous execution and incorporate a buffer space control mechanism. The details are in [8].

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 implied by elimination forests. Since we deal with a partitioned matrix, an element-wise elimination forest in Definition 1 needs to be clustered into a supernode-wise elimination forest. We call the new forest as a **supernodal elimination forest**.

Definition 4 *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 and only 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 element-wise elimination forest.*

As for the example in Figure 7(b), its supernodal elimination forest is depicted in Figure 8. The corresponding matrix is partitioned into 4×4 submatrices.

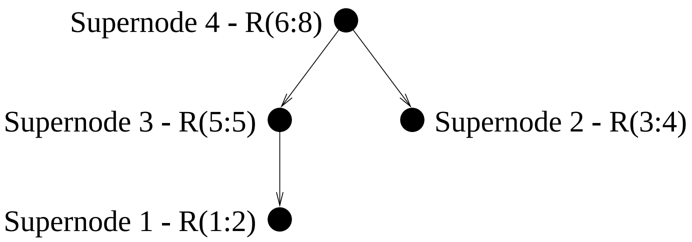

Figure 8: Supernodal elimination forest for the matrix in Figure 7(b)

.

A supernodal elimination forest can be generated efficiently in complexity $O(n)$ using the Theorem 3 below.

Theorem 3 *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 element-wise elimination forest.*

Proof: The "if" part is trivial according to Definition 4. 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 element-wise elimination forest. We show that j must be j_2 as follows. Since $R(j_1 : j_2)$ is a supernode, for every $u \in \{j_1, j_1 + 1, \cdots, j_2 - 1\}$, u's parent is $u + 1$ and it cannot be i. This concludes the proof. ∎

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

Theorem 4 *L part of supernode $R(j_1 : j_2)$ directly or indirectly updates L supernode $R(i_1 : i_2)$ if and only if $R(i_1 : i_2)$ is an ancestor of supernode $R(j_1 : j_2)$.*

Proof: We know that L supernode $R(j_1 : j_2)$ directly or indirectly updates L supernode $R(i_1 : i_2)$ if and only if there exists an L_j ($j \in \{j_1, j_1 + 1, \cdots, j_2\}$) which directly or indirectly updates column L_i ($i \in \{i_1, i_1 + 1, \cdots, i_2\}$). Using Theorem 2 and Definition 4, this theorem is true. ∎

Our design for LU task scheduling using the above forest concept is different from the ones for Cholesky [1, 19] because pivoting and row interchanges complicate the flow control in LU. Using

Theorem 4, 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 the factor-ahead and elimination forest guided algorithms by determining 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 pivoting sequence at each elimination step k. We define the **stage overlapping degree** for updating tasks as

$$max\{|k-k'| \mid \text{There exist tasks } Update2D(k,*) \text{ and } Update2D(k',*) \text{ executed concurrently.}\}$$

It is proved in [8] that for the factor-ahead approach, the reachable overlapping degree for 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.

In our current design for the elimination forest guided approach, we enforce a constraint so that the above size of extra buffer space 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. Figure 9 shows the elimination forest guided approach based on the above strategy. It is obvious that allocating more buffers can relax this constraint and increase the degree of stage overlapping. Our current experimental study does not show a substantial advantage by doing that, however, more work is needed to investigate this issue under the memory constraint.

Example. Figure 10(a) and (b) are the factor-ahead and elimination forest guided schedules for the partitioned matrix in Figure 7(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 forest based schedule, $Factor(2)$ is executed in parallel with $Factor(1)$ because there is no dependence between them, implied by the forest in Figure 8. As a result, the length of this schedule is one unit shorter than the factor-ahead schedule.

5 Implementation with supernodal GEMM kernel

We examine how the computation-dominating part of the LU algorithm is efficiently implemented using the level of BLAS as high as possible. Computations in task $Update2D()$ involve the following

```
Let (my_rno, my_cno) be the 2D coordinates of this processor;
Let m be the smallest column block number assigned to 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) assigned to this processor.
  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
```

Figure 9: Supernode elimination forest guided 2D approach.

supernode block multiplication: $A_{i,j} = A_{i,j} - A_{i,k} * A_{k,j}$ where $k \le i$ and $k \le j$. The BLAS-3 GEMM routine [5] may not directly be applicable 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 $A_{i,k} * A_{k,j}$.

There could be several approaches to circumvent the above problem: One approach is to use the mixture of BLAS-1/2/3 routines. If $A_{i,k}$ and $A_{i,j}$ have the same row sparse structure, and $A_{k,j}$ 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 the space allocation and computation, and hence BLAS-3 GEMM can be employed more often. But 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 $A_{i,k} * A_{k,j}$. The result is stored in a temporal block. At the second step, we merge this temporal result block with $A_{i,j}$

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

Figure 10: Task schedules for matrix in Figure 7(b). $F()$ stands for $Factor()$, $S()$ stands for $ScaleSwap()$, $U()$ stands for $Update2D()$ and PC stands for Processor Column.

using subtraction. Figure 11 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 the BLAS-3 GEMM. A further optimization is to speedup 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 temporal 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.

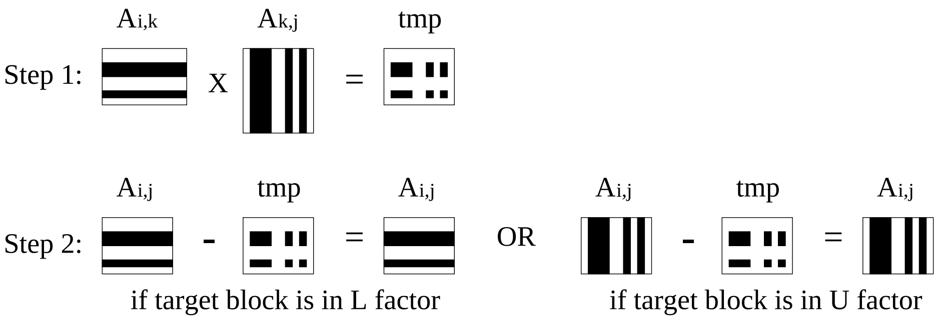

Figure 11: An illustration of Supernodal GEMM. Target block $A_{i,j}$ could be in the L factor or U factor.

Another issue is how to store and detect dense subrows and subcolumns. In our approach, for an L submatrix, subrows are stored in a consecutive space even their corresponding row numbers may not be consecutive. A bitmap scheme is used to indicate the supernode nonzero structure. A bit is set to 0 if the corresponding subrow is zero, and set to 1 otherwise. For example if an L submatrix contains only dense subrows, a bit is assigned for each subrow. Since we limit the supernode size no larger than 25 to fit the cache on T3E, we can use a 32-bit integer to store the bitmap of each submatrix, and can determine if a subrow is dense efficiently using a single logical "and" operation. The strategy for a U submatrix is the same except in a subcolumn-oriented fashion.

6 Experimental studies on Cray T3E

In this section, most of the experiments are conducted on T3E at San Diego Supercomputing Center (SDSC) unless explicitly stated. 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 [22]. We have observed that when block size is 25, double-precision GEMM achieves 388MFLOPS while double precision GEMV reaches 255MFLOPS. We have used block size 25 in our experiments. We recently obtained an 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 of experiments to show the performance improvement on an upgraded machine.

We will first report the overall sequential and parallel performance of our new code compared to SuperLU and our previous design. Then we measure the effectiveness of the proposed optimization strategies. In calculating the MFLOPS achieved by our parallel algorithms, *we do not include extra floating point operations introduced by the static fill-in overestimation.* The achieved MFLOPS is computed by using the following formula:

$$\text{Achieved MFLOPS} = \frac{\text{Operation count obtained from SuperLU}}{\text{Elapsed time of our algorithm on T3E}}.$$

Table 1 shows the statistics of the tested matrices. Column 2 is the order of the matrix and column 3 is the number of nonzeros before symbolic factorization. We have also listed the total number of factor entries divided by $|A|$ in SuperLU, S^+ and Cholesky factorization of A^TA for these matrices. And the comparison is shown in column 4, 5 and 6. The result shows that the overestimation in S^+ usually leads to less than 50% extra nonzeros than SuperLU does. But the A^TA approach overestimates substantially more nonzeros, which also indicates that the elimination tree of A^TA introduces too many false dependency edges. All matrices are ordered using the minimum degree algorithm and the permutation algorithm for zero-free diagonal [6]. In subsection 6.3, we will also report performance of S^+ for circuit simulation matrices.

6.1 Overall code performance

Our previous study [8, 10] shows that even with the introduction of extra nonzero elements by static symbolic factorization, the performance of the S^* sequential code can still be competitive to SuperLU because we are able to use more BLAS-3 operations. Table 2 shows new code S^+ can actually be faster than the SuperLU because of using new supernode partitioning and matrix

			factor entries/$|A|$		
Matrix	Order	$|A|$	SuperLU	S^+	A^TA
sherman5	3312	20793	12.03	15.70	20.42
lnsp3937	3937	25407	17.87	27.33	36.76
lns3937	3937	25407	18.07	27.92	37.21
sherman3	5005	20033	22.13	31.20	39.24
jpwh991	991	6027	23.55	34.02	42.57
orsreg1	2205	14133	29.34	41.44	52.19
saylr4	3564	22316	30.01	44.19	56.40
goodwin	7320	324772	9.63	10.80	16.00
e40r0100	17281	553562	14.76	17.32	26.48
raefsky4	19779	1316789	20.36	28.06	35.68
inaccura	16146	1015156	9.79	12.21	16.47
af23560	23560	460598	30.39	44.39	57.40
fidap011	16614	1091362	23.36	24.55	31.21
vavasis3	41092	1683902	29.21	32.03	38.75

Table 1: Testing matrices and their statistics.

multiplication strategies. The test matrices are selected from Table 1 that can be executed on a single T3E node. We include a test for factorizing a dense matrix to examine the algorithm performance on this extreme aspect. The improvement over SuperLU for the dense case is the highest because our code can fully utilize BLAS-3 for this case. We also compare the sequential performance of S^+ with our previous design S^* [10]. The performance improvement ratios vary from 22% to 40%. For the dense case, there is no improvement because the results of partitioning and matrix multiplication between two versions are the same in this case.

For parallel performance, we compare our new code with the previous version [8] in Table 3 and the improvement ratio in terms of MFLOPS vary from 16% to 116%, in average more than 50%. Table 4 shows the absolute performance of the S^+ on an LBL's Cray T3E machine with 450MHz CPU. The highest performance reached is 11.04GFLOPS, while for the same matrix, 8.44GFLOPS is reached on 300MHz T3E.

6.2 Effectiveness of the proposed optimization strategies

Elimination forest guided partitioning and amalgamation. Our new strategies for supernode partitioning with amalgamation cluster columns and rows simultaneously using structural containment information implied by an elimination forest. Our previous design S^* [10] does not consider the bounding of nonzeros in the U part. We compare 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 12 and an average of 20% improvement is obtained. The ratio for matrix "af23560" is not substantial because this matrix is very sparse and even new partitioning/amalgamation strategy

Matrix	Sequential S^+		SuperLU		Sequential S^*		Exec. Time Ratio	
	Time	Mflops	Time	Mflops	Time	Mflops	S^+/SuperLU	S^+/S^*
sherman5	0.65	38.88	0.78	32.4	0.94	26.9	0.83	0.69
lnsp3937	1.48	28.52	1.73	24.4	2.0	21.1	0.86	0.74
lns3937	1.58	28.30	1.84	24.3	2.19	20.4	0.86	0.72
sherman3	1.56	39.52	1.68	36.7	2.03	30.4	0.93	0.77
jpwh991	0.52	33.38	0.56	31.0	0.69	25.2	0.93	0.75
orsreg1	1.60	39.11	1.53	40.9	2.04	30.7	1.05	0.78
saylr4	2.67	40.10	2.69	39.8	3.53	30.3	0.99	0.76
goodwin	10.26	65.28	-	-	17.0	39.4	-	0.60
dense1000	4.04	165.0	8.39	79.4	4.04	165.0	0.48	1.00

Table 2: Sequential performance on a 300MHz Cray T3E node. Symbol "-" implies the data is not available due to insufficient memory.

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	813.4	958.4	1519.0	1864.8	2651.9	3303.6	4505.5	5640.4	6878.1	8441.9

Table 3: MFLOPS performance of S^+ and S^* on 300MHz Cray T3E.

can not produce large supernodes.

Effectiveness of supernodal GEMM. We assess the gain due to the introduction of the supernodal GEMM operation. We compare S^+ with a modified version using a mixed approach which mixes BLAS-1/2/3 as described in Section 5. We don't compare with the approach that treats all nonzero blocks 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 13. 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%.

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 factor-ahead and basic approaches described in Section 4. Compared to the basic approach, the improvement

Matrix	P=8		P=16		P=32		P=64		P=128	
	Time	Mflops	Time	Mflops	Time	Mflops	Time	Mflops	Time	Mflops
goodwin	1.21	553.5	0.82	816.8	0.69	970.6	0.68	984.9	0.67	999.6
e40r0100	4.06	611.3	2.50	992.7	1.87	1327.2	1.65	1504.1	1.59	1560.9
raefsky4	38.62	804.2	20.61	1507.0	11.54	2691.5	6.80	4567.6	4.55	6826.0
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	1423.6	33.68	2649.3	19.26	4632.9	11.75	7594.0	8.08	11043.5

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

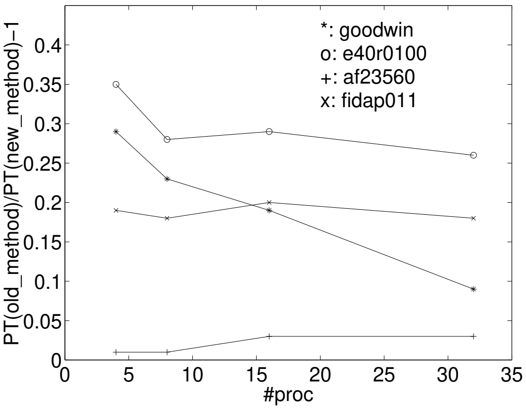

Figure 12: Performance improvement by using new supernode partitioning/amalgamation strategy.

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.

6.3 Performance on circuit simulation matrices

We recently obtained a few matrices from circuit simulation in Texas Instruments [23], for which the static factorization may generate many extra fill-ins. We chose three of them which are large enough for parallel test and ran them using S^+ on 450MHz Cray T3E. Table 6 shows that static factorization does produce a large number of fill-ins for these matrices (up to 3 times higher than dynamic factorization using the same matrix ordering). However, the experimental results in Table 7 demonstrate that S^+ still achieves decent MFLOPS in a large number of processors. Remember that we do not include extra floating point operations introduced by the static fill-in overestimation in calculating the MFLOPS achieved by our parallel algorithms. The achieved MFLOPS is computed as the operation count obtained from SuperLU divided by the parallel time of our algorithm on

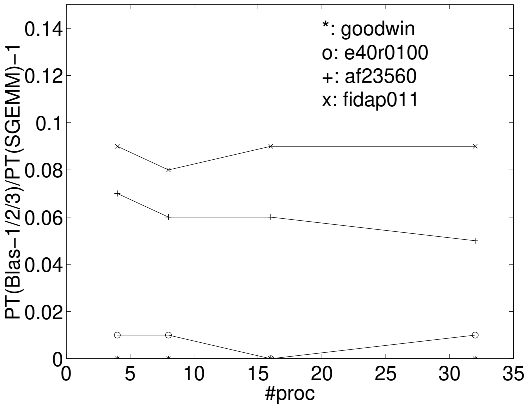

Figure 13: Performance improvement by using the supernodal GEMM.

Matrix	P=16		P=32		P=64		P=128	
	$\frac{Eforest}{Basic}$-1	$\frac{Eforest}{Fahead}$-1	$\frac{Eforest}{Basic}$-1	$\frac{Eforest}{Fahead}$-1	$\frac{Eforest}{Basic}$-1	$\frac{Eforest}{Fahead}$-1	$\frac{Eforest}{Basic}$-1	$\frac{Eforest}{Fahead}$-1
goodwin	41%	8%	35%	12%	19%	10%	21%	14%
e40r0100	38%	15%	40%	17%	30%	12%	27%	15%
raefsky4	21%	7%	21%	10%	34%	11%	34%	13%
inaccura	21%	7%	28%	13%	26%	9%	27%	13%
af23560	31%	10%	37%	15%	32%	10%	30%	13%
fidap011	24%	8%	28%	12%	36%	11%	38%	15%
vavasis3	17%	3%	16%	6%	31%	8%	28%	12%

Table 5: Performance improvement by using the elimination forest guided approach.

T3E. For the memory requirement, even though running S^+ on a single node needs more space for these matrices due to static factorization, such a demand per processor decreases linearly when more processors are involved. This is because our design of buffering in Section 4 makes sure the space complexity per processor is around S_1/p, where S_1 is the sequential space complexity and p is the number of processors.

7 Concluding remarks

Our experiments show that properly using elimination forests can guide us 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^+ and S^* can deliver high performance for large sparse matrices with reasonable space cost. Static symbolic factorization may create too many fill-ins for some matrices, in which case S^+ can still achieve good performance on a large number of processors.

Matrix	Order	$\|A\|$	factor entries/$\|A\|$		
			SuperLU	S^+	A^TA
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

Table 6: Circuit simulation testing matrices and their statistics.

Matrix	P=8		P=16		P=32		P=64		P=128	
	Time	Mflops	Time	Mflops	Time	Mflops	Time	Mflops	Time	Mflops
TIa	0.64	323.7	0.40	517.9	0.31	668.3	0.28	739.9	0.26	796.8
TId	1.98	298.6	1.15	514.1	0.80	739.0	0.62	953.6	0.54	1094.8
TIb	47.88	148.7	25.05	284.2	14.05	506.7	7.82	910.3	4.98	1429.5

Table 7: Performance of S^+ for circuit simulation matrices on 450MHz Cray T3E. All times are in seconds.

Therefore our approach is applicable to a large range of problems using a simple ordering strategy (minimum degree ordering). It might be possible to use different matrix reordering to reduce overestimation ratios and more studies are needed on this issue.

Acknowledgment

This work is supported by NSF CDA-9529418, NSF CAREER CCR-9702640, and DARPA DABT-63-93-C-0064 through the Rutgers HPCD project.

We would like to thank Cong Fu for continuous help on this project, Horst Simon for providing access to a Cray T3E at 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, Esmond Ng for helpful discussions, Apostolos Gerasoulis and the anonymous referees for their valuable comments.

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*, 1:10–30, 1987.

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

[3] 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.

[4] 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.

[5] 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.

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

[7] 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.

[8] 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.

[9] C. Fu and T. Yang. Run-time Compilation for Parallel Sparse Matrix Computations. In *Proceedings of ACM International Conference on Supercomputing*, pages 237–244, Philadelphia, May 1996.

[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] 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*, pages 419–430, Manchester, July 1994.

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

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

[15] A. Gupta, G. Karypis, and V. Kumar. Highly Scalable Parallel Algorithms for Sparse Matrix Factorization. *IEEE Transactions on Parallel and Distributed Systems*, 8(5), 1995.

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

[17] 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.

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

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

[20] E. Rothberg and R. Schreiber. Improved Load Distribution in Parallel Sparse Cholesky Factorization. In *Proc. of Supercomputing'94*, pages 783–792, November 1994.

[21] R. Schreiber. *Scalability of Sparse Direct Solvers*, volume 56 of *Graph Theory and Sparse Matrix Computation (Edited by Alan George and John R. Gilbert and Joseph W.H. Liu)*, pages 191–209. Springer-Verlag, New York, 1993.

[22] 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*, August 1996.

[23] A. Sherman and V. Gupta. Personal Communication, 1998.