Efficient Sparse Gaussian Elimination with Lazy Space Allocation

by

Bin Jiang

A dissertation submitted in partial satisfaction of the
requirements for the degree of
Master of Science

in

Computer Science

in the

GRADUATE DIVISION
of the
UNIVERSITY of CALIFORNIA at SANTA BARBARA

Committee in charge:

Professor Tao Yang, Chair
Professor Linda Petzold
Professor John Bruch

June 1999

Abstract

Efficient Sparse Gaussian Elimination with Lazy Space Allocation

by

Bin Jiang

Master of Science in Computer Science

University of California at Santa Barbara

Professor Tao Yang, Chair

A parallel algorithm is implemented for sparse Gaussian elimination on distributed memory machines. At First, we utilize the minimum degree ordering algorithm and transversal algorithm to reorder the columns and rows of the matrix. Next, we implement the LU factorization of the reordered matrix by combining various techniques, such as static symbolic factorization, 2D supernode partitioning, asynchronous computation scheduling and the new lazy space allocation strategy. This lazy space allocation strategy can effectively control memory usage, especially when static symbolic factorization overestimates fill-ins excessively. Our experiments show that the new LU code using this strategy has sequential time and space cost competitive with SuperLU, and can deliver up to 10 GFLOPS when running on 128 Cray 450Mhz T3E nodes. At last, we implement the triangle solve phase of Gaussian Elimination by using the same data mapping scheme as in LU factorization. The software implementing this GE algorithm is released to public and can be ported to Cray T3E and SGI Origin 2000 systems.

Contents

Acknowledgements

I would like to thank every one who has helped me during my graduate study at UCSB. In particular, I would like to thank my advisor Tao Yang, who has always been available for discussions and insightful comments, so I could pursue research in the right direction. I thank Prof. Linda Petzold and Prof. John Bruch for serving on my thesis committee. I also own thanks to Kai Shen and Steve Richman, who have been working with me for a long time and helped me a lot for this paper, and Xiangmin Jiao and Cong Fu, who are the past members of our project but they still provided me with some good advice for this thesis. Finally I would like to thank my wife Jinhua for her support and much good advise towards my future career.

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

Chapter 1

Introduction

A time and space efficient parallelization for sparse Gaussian elimination with pivoting is the key to solving large sparse non-symmetric linear systems. Such linear systems arise from a wide range of areas, such as computational fluid dynamics, structural engineering and device simulation etc., and many of the systems involve tens of thousands of unknowns. Solving these large linear systems may involve tens of billions of floating point operations and require gigabytes of memory. A high performance sequential computer may not be powerful enough to solve the problems. It is very important to design a parallel algorithm which can aggregate computational power and memory of multiprocessors to efficiently solve large sparse linear systems.

As far as we know, there is no published result for parallel sparse LU on current commercially available distributed memory machines such as Cray-T3D, Intel Paragon, IBM SP/2, TMC CM-5 and Meiko CS-2. One difficulty in the parallelization of sparse LU on these machines is how to utilize the sophisticated uni-processor architecture. The design of a sequential algorithm must take advantage of caching, which makes some previously proposed techniques less effective. On the other hand, a parallel implementation must utilize the fast communication mechanisms available on these machines. It is easy to get speedups by

comparing a parallel code to a sequential code which does not fully exploit the uni-processor capability, but it is not as easy to parallelize a highly optimized sequential code. One such sequential code is SuperLU [7] which uses a supernode approach to conduct sequential sparse LU with column partial pivoting. The supernode partitioning makes it possible to perform most of the numerical updates using BLAS-2 level dense matrix-vector multiplications, and therefore to better exploit memory hierarchies. They perform symbolic factorization and generate supernodes on the fly as the factorization proceeds. Their code delivers impressive performance and is among the best sequential codes for sparse LU with partial pivoting [7, 3]. However it is challenging to parallelize their code to get scalable performance and so far we have not seen any published results on the parallelization of their method on distributed memory machines.

In [15] we presented a novel approach that considers three key optimization strategies together in parallelizing the sparse LU algorithm: 1) adopt a static symbolic factorization scheme to eliminate the data structure variation caused by dynamic pivoting; 2) identify data regularity from the sparse structure obtained by the symbolic factorization scheme 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 to exploit irregular parallelism. We observe that on most current commodity processors with memory hierarchies, a highly optimized BLAS-3 subroutine usually outperforms a BLAS-2 subroutine in implementing the same numerical operations [6, 8]. We can afford to introduce some extra BLAS-3 operations in re-designing the LU algorithm so that the new algorithm is easily parallelized but the sequential performance of this code is still competitive to the current best sequential code. We use the static symbolic factorization technique first proposed in [16, 17] to predict the worst possible structures of the L and U factors without knowing the actual numerical values, then we develop a non-symmetric L/U supernode partitioning technique to identify the dense structures in both the L and U factors, and maximize the use of BLAS-3 level

subroutines (matrix-matrix multiplication) for these dense structures. We also incorporate a supernode amalgamation technique to increase the granularity of the computation.

Recently [23] we have further studied the properties of elimination forests to guide supernode partitioning/amalgamation and execution scheduling. The new code with 2D mapping, called S^+, effectively clusters dense structures without introducing too many zeros in the BLAS computation, and uses supernodal matrix multiplication to retain the BLAS-3 level efficiency and avoid unnecessary arithmetic operations. The experiments show that S^+ improves our previous code substantially and can achieve up to 11.04GFLOPS on 128 Cray 450MHz T3E nodes.

Our previous evaluation shows that for most of the tested matrices, 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 static symbolic factorization creates too many fill-ins and our previous solution does not provide a smooth adaptation in handling such cases. For these cases, we find that the prediction can contain a significant number of fill-ins that remain zero throughout the numerical factorization. This indicates that space allocated to those fill-ins is unnecessary.

Thus our first space-saving strategy is to delay the space allocation decision and acquire memory only when a submatrix block becomes truly nonzero during numerical computation. Such a dynamic space allocation strategy can lead to a relatively small space requirement even if static factorization excessively overpredicts nonzero fill-ins.

Another strategy we have proposed is to examine if space recycling for some nonzero submatrices is possible since a nonzero submatrix may become zero during numerical factorization due to pivoting and number subtraction. This has the potential to save significantly more space since the early identification of zero

blocks prevents their propagation in the update phase of the factorization.

Since our S^+ method with lazy space allocation shows a great deal of advantages over other software for the sparse matrix LU factorization on distributed memory machines, such as Cray T3E, SGI Origin 2000, we implemented this S^+ software utilizing all the methods we will talk about in this thesis to the solve sparse matrix equation $AX = B$ on the distributed memory machines. This software is implemented by the widely used Message Passing Interface(MPI). Since MPI code can be run under almost all parallel system, our software can be applicable to all machines where MPI library is installed.

Generally speaking, sparse Gaussian elimination consists of three steps: **analyze**, **factorize** and **solve** [2, 10], compared to the dense elimination which has only the last two steps. The concepts of factorize and solve of the sparse Gaussian elimination are the same as those of the dense elimination, whereas there exist some minor differences between them. The rest of this paper is organized as follows. Chapter 2 concerns the analyze phase of our S^+, and presents the ordering and a new transversal technique for the matrices before Gaussian elimination. Chapter 3 presents the factorize phase of our S^+ code and introduces two Space Optimization methods for S^+. The experimental results on a Sun Ultra-1 Sparcstation and Cray T3E show that our new code uses less space than our old code and is faster than the SuperLU method. Chapter 4 concerns the solve phase and presents a parallel forward substitution algorithm and a parallel backward substitution algorithm for the triangular solving with the same data mapping scheme to avoid data shuffling between processors. Chapter 5 introduces the S^+ software implemented by MPI. A comparison of time performance of MPI code with SHMEM code is also provided, showing that MPI code is comparable to SHMEM code under Cray T3E and therefore can be used in practice. Chapter 6 concludes the paper and proposes some future research directions.

Chapter 2

Ordering and Transversal of Sparse Matrix

The analyze phase is the preprocessing step for sparse Gaussian elimination. The key features of the analyze phase are to preserve sparsity by using some ordering algorithms.

Ordering is an important issue in sparse matrix computation, because different ordering of equations and variables can have a significant impact on fill, which determines the memory requirements, number of floating-point operations and caching performance.

To find an optimal ordering for a given matrix is an NP-hard problem [24]. Some heuristic ordering algorithms have been extensively discussed in [18], and better algorithms are still under investigation [5].

In this thesis, we use the column minimum degree ordering implemented by J.W.Liu [18]. It has been verified that the minimum column degree ordering can reduce the nonzero fill-ins during the step of symbolic factorization in our S^+.

2.1 Transversal Algorithm by Duff

However, after the ordering of columns, the diagonal of the matrix may contain many zeros due to the permutation of columns. This will introduce extra nonzeros into the structure of the L and U factors and bring more burden to the LU factorization, because Symbolic factorization determines the *upper bound* of the structures of the L and U factors without actually computing the factors numerically as follows:

"At each step k $(1 \leq k < n)$, each row $i \geq k$ which has a nonzero element in column k consists *candidate pivot rows* for row k. As the static symbolic factorization proceeds, at step k the nonzero structures of *each* candidate pivot row for k is replaced by the union of the structures of all these candidate pivot rows and the kth row(no matter whether or not a_{kk} is zero) except the first $k-1$ columns."

Since the kth row will always be taken into the union at step k even though its diagonal element a_{kk} is zero, but this row won't be taken into the union at step k if it is not the kth row based on the above principle, therefore we should keep all (or as many as we can) the diagonal elements as nonzeros, then reduce the possibility of rows with zero diagonal being taken into the union while in fact it won't be considered for the union if it is not the kth row at step k. In this way we can decrease the number of candidate rows at each step.

Therefore a transversal algorithm to transverse the rows to produce a zero-free diagonal is necessary for the matrix being ordered by minimum degree ordering. The most popular transversal algorithm is Duff's algorithm [9] which ensures the transverse matrix will have a zero-free diagonal.

First let's describe the basic techniques of the algorithm, making use of some terminology from graph theory.

The transversal is constructed in n major steps, after the kth of which we have a transversal for a submatrix of order k. After the kth step, we associated

with the matrix an unconventional directed graph(which usually changes from step to step). Each vertex of the graph corresponds to a row of the matrix, and there is an edge from vertex i_0 to vertex i_1 if there exists a column of the matrix, j_1 say, such that nonzero $(i_1,\ j_1)$ is a current transversal element and element $(i_0,\ j_1)$ is nonzero. We say we can reach vertex i_1 from vertex i_0 and define a path to be a sequence of edges of this kind. It is helpful to consider a path, from i_0 to i_k, say, as a sequence of nonzero $(i_0, j_1), (i_1, j_2), \ldots, (i_{k-1}, j_k)$ where the present transversal includes the nonzeros $(i_1, j_1), (i_2, j_2), \ldots, (i_k, j_k)$. Now if there is a nonzero in position (i_k, j_{k+1}), and if no nonzero in row i_0 or column j_{k+1} is currently on the transversal, then the length of the transversal can be increased by 1 by removing nonzeros $(i_r, j_r), r = 1, \ldots, k$, from the transversal and adding nonzeros $(i_r, j_{r+1}), r = 0, 1, \ldots, k$ to it. In Figure 2.1, we illustrate this reassignment chain on the matrix representation.

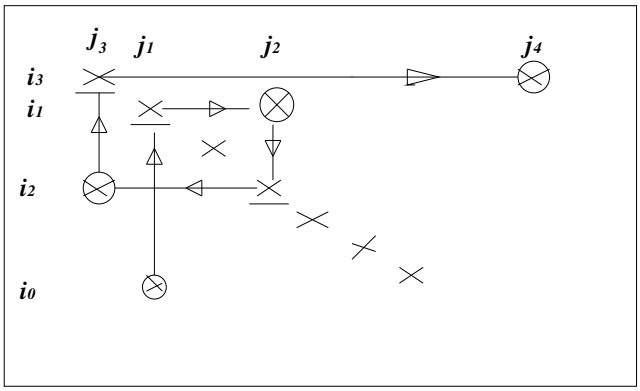

Figure 2.1: Reassignment Chain

The reassignment is shown by directed lines in Figure 2.1, the vertical lines from nonzeros (i_r, j_{r+1}) to (i_r, j_r), and horizontal lines from (i_r, j_r) to (i_r, j_{r+1}). The reassignment corresponds to replacing the three underlined transversal elements by the circled nonzeros.

We use a depth first search technique to find the reassignment chain. We search edges from the current vertex and add to our path the first vertex en-

countered that we have not revisited. This becomes the current vertex and we proceed from it as before. If all the vertices that can be reached from the current one at the end of the path are already visited, we retrace our steps to the vertex added to the path immediately before this present one, make that the current vertex, and proceed as before. We define our edges, as before, to be the form of (i_1, i_2) where (i_1, j_2), say, is a nonzero and (i_2, j_2) a present assignment. We start from any unassigned vertex(row) i_0 and trace a path using a DFS technique until a vertex i_k is reached where the path terminates because nonzero (i_k, j_{k+1}) exists and j_{k+1} is an unassigned column.

In practice this algorithm is very inefficient because the DFS scheme does not specify which unvisited vertex reached from the present current vertex should be added to the path and such a choice could be quite critical.

The complexity of the algorithm is $O(n\tau)$, where n is the order of the matrix and τ is the number of nonzeros of the matrix. Therefore, for a large matrix, the transversal algorithm is very time consuming(You will see it from Table 2.1), and it will be very inefficient to use the original Duff's algorithm to handle transversal in our code. In Section 2.2 we propose a new approximation transversal algorithm which is much faster than Duff's algorithm.

2.2 Approximate Transversal Algorithm in S^+

Notice that the transversal algorithm is used to eliminate zeros on the diagonal of a matrix, to produce a sparser L and U structure in the step of symbolic factorization. The optimal transversal algorithm is of order $O(n\tau)$. We implemented an approximate algorithm which is of order only $O(\tau)$ and will produce an almost zero-free diagonal for the matrix(the proportion of nonzeros on the diagonal is less than 1%) and therefore the symbolic factorization will produce almost the same sparser L and U structure for symbolic factorization but the time saving for transversal is significant.

We propose a new approximation transversal algorithm as follows:

Suppose the matrix has n rows. Each row contains some nonzero elements. We use the array LeftOver[i] to denote the number of nonzeros in row i, where $1 \leq i \leq n$.

At step 1, we go through all the nonzero elements of the 1st column. For each element whose row number is i, the corresponding LeftOver[i] stands for the number of nonzeros in row i. We choose the row whose LeftOver[i] is minimum and then interchange row 1 with row i. At the same time, we decrease all the LeftOver[i] by 1 for all nonzeros i in the 1st column and mark this row i so that it won't be picked up in later steps. The row whose LeftOver[i] is minimum at this step has the least possibility to be picked up as a candidate row for exchange in later steps, therefore we choose it to be the candidate at this step. We decrease LeftOver[i] by 1 for those i appearing in the first column since LeftOver[i] stands for the remaining nonzero elements in row i at present. At each step k, we only care about how many nonzero leftovers a row still owns from column k to column n, therefore the LeftOver value will be decreased by 1 at the end of the current step.

Generally, at step k, we go through all nonzeros in column k which are not marked(The marked rows have been picked up as candidate row for prior steps so they can not be used here). Choose the row i whose LeftOver[i] is minimum. Then we interchange row i with row k and mark row i correspondingly. In this way, we can transverse all the rows of the matrix and will make the matrix almost zero diagonal free by interchange rows according to the LeftOver[i] value.

It is easy to see that we avoid the depth first search algorithm which is very time consuming and inefficient. We will see from the following table that our approximate algorithm is very fast, but the trade-off is that there may exist some zeros on the diagonal but the percentage is very very small, less than 1%, therefore it won't affect the time performance for symbolic factorization.

We can prove that the complexity of the above algorithm is O(τ), where τ

is the number of nonzeros of the matrix. Recalling the complexity of Duff's algorithm is $O(n\tau)$, we can see, for a matrix whose row number n is large, the time saving of our approximate algorithm is significant over Duff's Algorithm from a theoretical point of view.

Table 2.1 compares the time used by the original Duff method applied to the benchmark matrices after mmd ordering and that of our approximation algorithm for the same matrices. The time for mmd ordering for those matrices is also provided for clarity. We can see that the time of our algorithm is less than 1% of that spent by the Duff method on average, and can be ignored compared with the time for mmd ordering. The last column shows the proportion of the nonzero elements on the diagonal after using mmd and the approximate transversal. It is usually less than 0.5% and will bring no negative effect for the symbolic factorization.

Table 2.1: Comparison of Time Performance of Duff Transversal Method and our Approximate method. Time is in seconds.

Matrix	MMD	Duff method	Approximate method	$\frac{\text{zeros on the Diagonal}}{\text{Matrix Order}}$
OLAF1	9.82	98.18	0.25	220/16146
ex11	12.90	121.91	0.29	197/16614
goodwin	2.82	1.783	0.09	0/7320
jpwh991	0.087	0.114	0.003	12/991
memplus	111.98	1.88	0.09	78/17758
orsreg1	0.43	0.41	0.69	64/2205
raefsky4	13.91	178.65	0.359	343/19779
saylr4	0.47	5.53	0.02	76/3564
sherman3	0.25	0.66	0.02	62/5005
sherman5	0.15	0.27	0.01	23/3312

Chapter 3

Sparse LU factorization with Lazy Allocation

3.1 Strategies of S^+ Implementation

3.1.1 Static symbolic factorization.

After ordering, the data structures are prepared for runtime execution. The data structures of sparse computation are much more complicated than those of dense computation. One important data structure of sparse code is *matrix structure*, which stores the sparsity pattern of the matrix. In Gaussian elimination with partial pivoting, the matrix structures of the factors are unknown before factorization, because the pivoting sequence is unknown until the factorize phase. A straightforward method is to dynamically construct the data structure during runtime. This method keeps track of the fill during runtime and allocates space for new fill dynamically. The method has the advantage of precise manipulation of fill. However the disadvantages of this method are that it introduces high overhead of memory management, one has to parallelize *symbolic factorization* and *numerical factorization* interleavingly, and it is very difficult to

implement on a distributed memory architecture. Another method is to do symbolic factorization statically, i.e., to predict all possible fill-ins and allocate space for them before actual numerical computation. This method overestimates fill-ins and allocates memory space to avoid dynamic memory management. It has been shown that the static approach is competitive with the dynamic approach in terms of performance as well as memory requirements for a broad range of linear systems [7, 11, 13, 17] and it is much easier to be parallelized on distributed memory machines.

Static symbolic factorization is proposed in [17] 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."

It is easy to see that this algorithm guarantees that L and U structures are contained in the resulting matrix structure regardless of pivoting sequences. The symbolic factorization process of the sample matrix in Figure 3.1 is shown in Figure 3.2.

Using an efficient implementation of the symbolic factorization algorithm [19], 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 [7, 20], provides more accurate data structure prediction on the fly, but it is challenging to parallelize SuperLU with low runtime control overhead on distributed memory

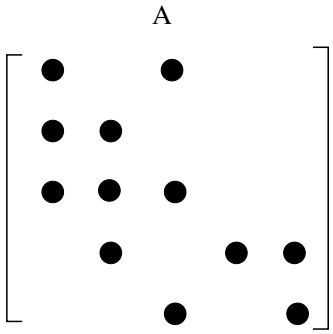

Figure 3.1: A sample sparse matrix.

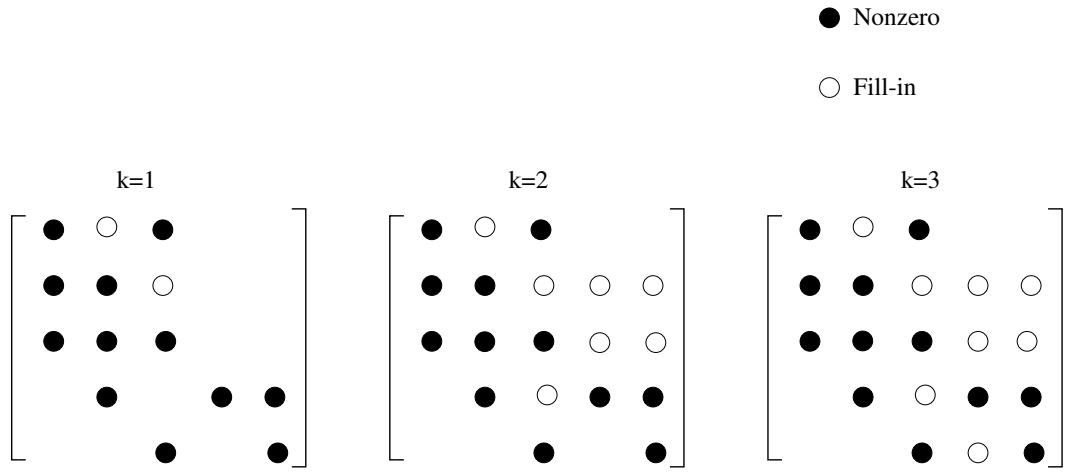

Figure 3.2: The first 3 steps of the symbolic factorization on a sample 5×5 sparse matrix. The structure remains unchanged at step 4.

machines. In [14, 15], 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 a 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.

3.1.2 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 [9]. 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 3.3 illustrates a sparse matrix after symbolic factorization and its elimination forest.

The following property 1 below demonstrates the structural properties of an elimination forest.

Property 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 $U_k - \{k, k+1, \cdots, j-1\} \subseteq U_j$.

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**

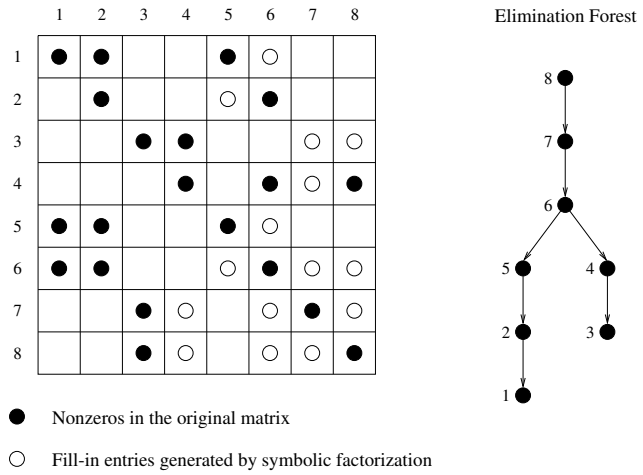

Figure 3.3: A sparse matrix and its elimination forest.

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 $1 \leq q \leq p-1$.

Property 2 below identifies the dependency information in the elimination forest.

Property 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.

Property 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. Property 2 indicates dependency information in the numerical elimination, which can guide our parallel scheduling of asynchronous parallelism.

3.1.3 2D L/U supernode partitioning and amalgamation.

Given a nonsymmetric matrix A after symbolic factorization, in [15] 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 Property 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 [15] and hence the structures of U_i and U_{i-1} are identical. Figure 3.4(a) illustrates supernode partitioning of the sparse matrix in Figure 3.3. 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.

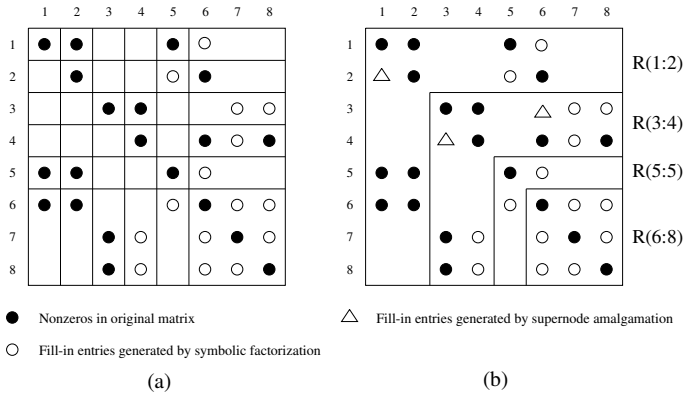

Figure 3.4: (a) Supernode partitioning for the matrix in Figure 3.3; (b) The result of supernode amalgamation.

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. [22]), 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 [15], 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 Property 1, can be used to bound the nonzero structure of a relaxed L/U supernode.

Corollary 2 *If for each i where $s+1 \leq i \leq t$, 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 $s+1 \leq i \leq t$, 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.

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.

Figure 3.4(b) illustrates the result of supernode amalgamation for the sparse matrix in Figure 3.3. Condition $z \leq 30\%$ is applied during the amalgamation. There are four relaxed L/U supernodes: $R(1:2)$, $R(3:4)$, $R(5:5)$, and $R(6:8)$.

3.1.4 2D data mapping and asynchronous parallelism exploitation.

Given an $n \times n$ matrix A, assume that after the matrix partitioning it has $N \times N$ submatrix blocks. Let $A_{i,j}$ denote a submatrix of A with row block index i and column block index j. We use 2D block-cyclic mapping: processors are viewed as a 2D grid, and a column block of A is assigned to a column of processors. 2D sparse LU Factorization is more scalable than the 1D data mapping [12]. However 2D mapping introduces more overhead for pivoting and row swapping. Each column block k is associated with two types of tasks: $Factor(k)$ and $Update(k,j)$ for $1 \leq k < j \leq N$. 1) Task $Factor(k)$ factorizes all the columns in the k-th column block, including finding the pivoting sequence associated with those columns and updating the lower triangular portion of column block k. The pivoting sequence is held until the factorization of the k-th column block is completed. Then the pivoting sequence is applied to the rest of the matrix. This is called "delayed pivoting" [6]. 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 fac-

torized 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 3.5 outlines the partitioned LU factorization algorithm with partial pivoting.

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

Figure 3.5: Partitioned sparse LU factorization with partial pivoting.

In [23], we have proposed an asynchronous scheduling guided by the elimination forest. This strategy enables the parallelism exploitation among $Factor()$ tasks which used to be serialized by previous scheduling strategies.

3.2 Space Optimization Techniques

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

3.2.1 Delayed space allocation

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

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

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

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

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

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

- For relatively dense matrices, this strategy has almost no effect since almost all the blocks produced at the step of supernode partitioning contain at least

some nonzeros or will have some nonzeros during numerical factorization, the number of blocks of type 3 is very small. Thus lazy allocation won't save a lot of space for those matrices.

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

- Further savings can be reaped in another part of our code: numerical factorization. First of all, each $Factor$ task in numerical factorization needs to factorize one column block. And all zero blocks are unnecessary to get involved into this task. But as long as a block is recognized as a nonzero block in numerical factorization, S^+ still ran it even though it may be actually a zero block during numerical factorization. However, in $LazyS^+$ with delayed space allocation, those actually zero blocks are not allocated space throughout the numerical factorization and they will be treated as zero blocks without getting involved into the numerical factorization. The $Update$ tasks are the most time consuming part of numerical factorization. $Update(k, j)$ uses blocks $A_{i,k}$ and $A_{k,j}$ to update block $A_{i,j}$ for every $k < i \leq N$. If either $A_{i,k}$ or $A_{k,j}$ is a zero block, it is unnecessary to update block $A_{i,j}$ in this task (see Figure 3.6). However, S^+ code updated every $A_{i,j}$ if both $A_{i,k}$ and $A_{k,j}$ are recognized as nonzero blocks by symbolic factorization even though one of them is a zero block during numeric factorization. Therefore a lot of time was wasted in unnecessary updating. $LazyS^+$ with delayed space allocation gets rid of this shortcoming. It first checks block $A_{k,j}$. If it is a zero block, the whole $Update(k, j)$ task is skipped (see Figure 3.7(a)). Otherwise, it picks up the nonzero blocks $A_{i,k}$ in column k, and updates the corresponding blocks $A_{i,j}$ (see Figure 3.7(b)).

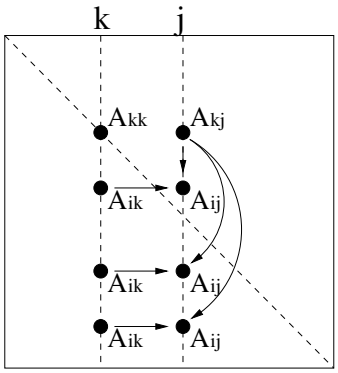

Figure 3.6: Illustration of $Update(k, j)$ task.

3.2.2 Space reclamation

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

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

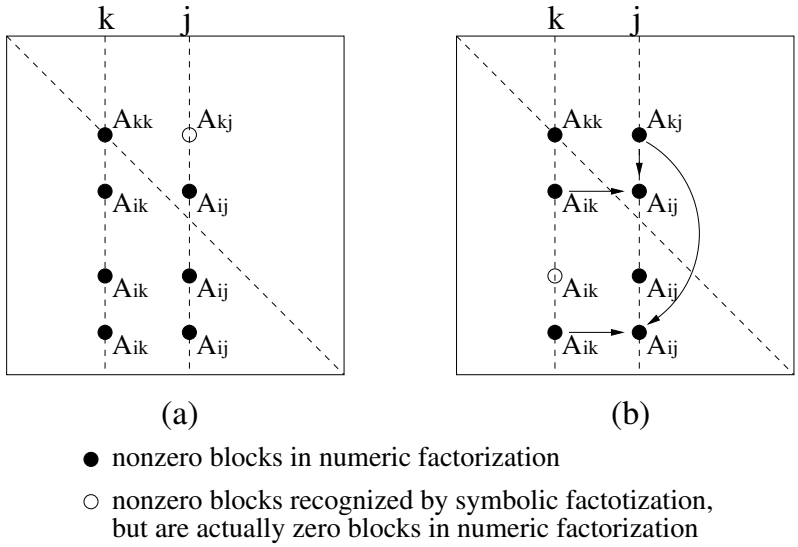

Figure 3.7: Illustration of $Update(k, j)$ task with delayed space allocation.

ginning of task $Update(k, j)$. If they have, their space will be deallocated and those blocks will also be excluded from future computation.

3.3 Performance Comparison of new S^+ with old code and SuperLU

3.3.1 Experimental Studies on Sequential Performance

The sequential machine we use is a SUN 167MHZ Ultra-1 with 320MB memory, 16KB L1 data cache and 512KB L2 cache. We have compared our sequential code with SuperLU, but not UMFPACK [4] because SuperLU has been shown competitive to UMFPACK [7]. The following benchmark matrices are used from various application domains: af23560, e40r0100, fidap011, goodwin, memplus, orsreg1, raefsky4, saylr4, sherman3, sherman5, TIa, TIb, TId and wang3. All matrices are ordered using the minimum degree algorithm. In computing gigaflop rates, we use operation counts reported by SuperLU for the tested matrices, which

excludes the extra computation introduced by static symbolic factorization.

Comparison of S^+, $LazyS^+$ and SuperLU

We compare the sequential performance of S^+, SuperLU and $LazyS^+$ on all the matrices. From Table 3.1, we can see the $LazyS^+$ code will run a little bit faster than the S^+ and SuperLU, while the space saving is not obvious. The reason is that these matrices do not have enough zero blocks for lazy allocation to be very advantageous. On average, for these non circuit simulation matrices, $LazyS^+$ will use 4.1% less space and 7.2% less time than S^+, while $LazyS^+$ will use 0.8% less space and 15.8% less time than SuperLU.

From Table 3.1, we can see obvious savings of space and time when $LazyS^+$ is compared with S^+ and SuperLU. The reason is twofold: first, these matrices contain a lot of zero blocks which aren't allocated space and involved in the computation when lazy allocation is used; second, S^+ and SuperLU cause paging on matrices TIb and wang3 due to the large amount of space needed. By excluding the paging effects, i.e. only considering the other three matrices when calculating the savings on time, $LazyS^+$ uses 41.7% less space and 62.5% less time than S^+ and it uses 9.6% less space and 60.7% less time than SuperLU.

Sensitiveness on block size limit

The above experiments use the block size limit 25. Table 3.2 shows the performance of $LazyS^+$ under different block size limits. For most matrices where fill-in overestimation is not excessive,, when we reduce this limit to 15, 10, and 5, changes in space saving are insignificant while processing time increases gradually due to degradation of caching performance.

For matrices with high fill-in overestimation, space saving is more effective when the block size is reduced. The reason is that when the block size becomes smaller, the probability of a block being zero block is higher. Therefore the lazy

Table 3.1: Sequential performance. A "-" implies the data is not available due to insufficient memory. Time is in seconds and space is in MBytes.

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

allocation strategy in $LazyS^+$ will become more effective.

Effectiveness of each individual lazy allocation strategy used in $LazyS^+$

We also conducted experiments concerning the effectiveness of each individual lazy allocation strategy. We are more interested in the circuit simulation matrices because they are the matrices in which $LazyS^+$ shows great advantages.

Table 3.3 shows the time and space performance on circuit simulation matrices by using S^+, $LazyS^+$ and $LazyS^+$ without space reclamation. We can see that the space reclamation plays a more important role than delayed allocation in the overall improvement. And on average, the delayed allocation saves 6.6% in time and 3.8% in space while the space reclamation saves 59.1% in time and 34.9% in space. Note that we exclude the paging effect in this calculation, i.e., only

Table 3.2: Sequential performance of $LazyS^+$ with different block size limits. Time is in seconds and space is in MBytes.

Matrix	25		15		10		5	
	Time	Space	Time	Space	Time	Space	Time	Space
sherman5	0.93	2.964	1.12	2.939	1.37	2.953	3.06	3.119
sherman3	2.20	5.536	2.79	5.545	3.52	5.600	7.92	5.988
orsreg1	1.95	4.730	2.23	4.558	2.31	4.308	3.34	4.128
saylr4	3.50	8.014	4.09	7.834	4.41	7.498	6.80	7.119
goodwin	14.91	28.995	0.89	29.168	28.02	29.725	79.10	32.760
e40r0100	54.78	78.568	76.45	79.170	102.39	80.373	334.93	88.968
raefsky4	606.72	285.920	809.11	279.931	997.88	278.565	2605.30	289.890
af23560	157.04	162.839	195.16	158.626	233.53	155.907	594.58	159.475
fidap011	360.62	219.208	513.05	220.791	688.81	225.337	2082.15	251.426
TIa	3.97	7.321	3.60	6.668	3.38	5.912	3.88	5.248
TId	11.00	19.655	8.29	15.090	7.50	12.641	7.77	12.410
memplus	44.21	68.467	29.19	62.752	19.35	56.590	17.98	41.666
TIb	73.97	107.711	62.29	79.032	70.74	66.089	185.74	84.377
wang3	645.15	347.505	627.20	310.836	602.84	282.690	952.76	270.136

counting matrices TIa, TId and memplus when calculating savings on time.

3.3.2 Experimental Studies on Parallel Performance

Our experiments on Cray T3E show that the parallel time performance of $LazyS^+$ is still competitive to S^+. It is shown in Table 3.4 that $LazyS^+$ can achieve 10.004 GFLOPS on matrix vavasis, which is not much less than the highest 11.04 GFLOPS achieved by S^+ on 128 450MHz T3E nodes. Table 3.5 is the performance on 300Mhz T3E nodes. Our study focuses on relatively large matrices.

Table 3.4 lists the parallel performance on 450MHz T3E nodes, and the performance data of $LazyS^+$ for some matrices is not available due to the insufficient

Table 3.3: Effectiveness of individual lazy strategy on circuit simulation matrices. Time is in second s and space is in MBytes.

Matrix	$LazyS^+$		$LazyS^+$ without space reclamation		S^+	
	Time	Space	Time	Space	Time	Space
sherman5	0.93	2.964	0.99	3.018	0.97	3.061
sherman3	2.20	5.536	2.36	5.657	2.33	5.713
orsreg1	1.95	4.730	2.33	5.038	2.37	5.077
saylr4	3.50	8.014	4.03	8.418	4.02	8.509
goodwin	14.91	28.995	15.42	29.070	15.02	29.192
e40r0100	54.78	78.568	55.07	78.761	52.87	79.086
raefsky4	606.72	285.920	672.02	300.094	658.89	303.617
af23560	157.04	162.839	163.45	167.710	159.62	170.166
fidap011	360.62	219.208	364.65	219.422	357.26	221.074
TIa	3.97	7.321	5.67	8.133	6.31	8.541
TId	11.00	19.655	27.73	27.682	30.03	29.647
memplus	44.21	68.467	338.22	138.633	344.64	138.218
TIb	73.97	107.711	1213.47	315.412	1325.29	341.418
wang3	645.15	347.505	1417.70	428.741	1431.91	430.817

CPU quota on this machine [1]. Nevertheless, data on 300MHz T3E nodes in Table 3.4 actually indicates that $LazyS^+$ is competitive with S^+ for these matrices. For the matrices with high fill-in overestimation ratios, we observe that $LazyS^+$ with dynamic space management is better than S^+. It is about 191% faster on 8 processors and 120% faster on 128 processors. Matrix wang3 can't run on T3E using S^+ since it produces too many fill-ins from static symbolic factorization. However $LazyS^+$ only allocates space if necessary, so considerable space is saved for a large amount of zero blocks.

As for other matrices, we can see from Table 3.5 that on 8 300Mhz processors $LazyS^+$ is about 1% slower than S^+ while on 128 processors, $LazyS^+$ is 7% slower than S^+. On average, $LazyS^+$ tends to become slower when the number of processors becomes larger. This is because the lazy allocation scheme introduces

[1] We will provide it when more computing resource is allocated.

Table 3.4: Time and MFLOPS performance of $LazyS^+$ and S^+ on 450MHz Cray T3E. A "-" implies the data is not available due to insufficient memory. A "*" implies the data is not available due to insufficient CPU quota on this machine. Time is in seconds.

Matrix	$LazyS^+$ P=8		S^+ P=8		$LazyS^+$ P=128		S^+ P=128	
	Time	MFLOPS	Time	MFLOPS	Time	MFLOPS	Time	MFLOPS
goodwin	*	*	1.21	553.5	*	*	0.67	999.6
e40r0100	*	*	4.06	611.3	*	*	1.59	1560.9
raefsky4	*	*	38.62	804.2	*	*	4.55	6826.0
af23560	*	*	10.57	602.1	*	*	2.80	2272.9
vavasis3	59.77	1492.9	62.68	1423.6	8.92	10004.0	8.08	11043.5
TIa	0.61	339.6	0.64	323.7	0.28	739.9	0.26	796.8
TId	2.10	281.5	1.98	298.6	0.59	1001.9	0.54	1094.8
TIb	12.81	555.7	47.88	148.7	2.83	2515.7	4.98	1429.5
memplus	8.28	0.2	-	-	1.82	1.0	-	-
wang3	74.69	194.9	-	-	9.04	1610.2	-	-

new overhead for dynamic memory management and for row and column broadcasts (blocks of the same L-column or U-row, now allocated in non-contiguous memory, can no longer be broadcasted as a unit). This new overhead affects critical paths, which dominate performance when parallelism is limited and the number of processors is large. This problem tends to become more serious when the number of processors is getting bigger.

We need to mention memplus, a special circuit simulation matrix. This matrix is excessively sparse: its order is 17758 but there are only 99147 nonzero elements. Therefore the operation count reported by SuperLU is not large, but, due to its special matrix structure (too sparse but still hard to conduct LU factorization), it LU factorization still require a large portion of time. Therefore its MFLOPS is much smaller than all the other test matrices.

Table 3.5: MFLOPS performance of S^+ and $LazyS^+$ on 300MHz Cray T3E.

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

Chapter 4

Parallel Triangular Solves

In this chapter, we discuss some issues about parallel triangular solves. In Section 4.1 we introduce some background information about triangular solves. In Section 4.2 we discuss how to implement parallel triangular solve by using the same data mapping scheme for the factorize phase so that data shuffling can be avoided. In Section 4.3 we present the time performance of parallel triangular Solves compared with that of factorization.

4.1 Introduction

Triangular solve requires many fewer floating point operations than sparse LU factorization. Less efforts have been made to parallelize it. However, parallelization of triangular solve is drawing more attention [7] because of the following reasons.

1. Limited memory resources of a sequential machine make it impossible to solve very large triangular systems. Memory may be the most important issue for some scientific and engineering computations.

2. After efficient parallelizations of sparse LU factorization have been developed, triangular solve may become the bottleneck of the whole process of

sparse Gaussian elimination.

3. The factor results of factorization are distributed on multiprocessors for space efficiency. Gathering the factor results to a single processor introduces high memory management and communication overhead.

4. For some applications, like multiple right-hand side systems and iterative refinement for linear systems [10], the triangular solves are executed multiple times by reusing the factor results. In these kinds of systems triangular solve might involve a comparable amount of computation as the factorization.

It is more difficult to parallelize sparse triangular solves than dense triangular solves. One obvious reason is that we have to exploit irregular parallelism by using asynchronous scheduling. More importantly, the data mapping of triangular solves is predetermined by sparse LU factorization. We can not select the most preferable mapping scheme for triangular solve because it may require global data movement. It has been shown that 2-D data mapping scheme is more scalable than 1-D data mapping for sparse LU factorization with partial pivoting, but 2-D data mapping also makes the parallelization of triangular solves more complicated. Lastly, because the L factor of sparse Gaussian elimination is stored differently from that of dense elimination, the dependence of forward substitution is more complicated in sparse elimination. In this chapter, we design the forward substitution algorithm to solve $Ly = b$ and the backward substitution algorithm to solve $Ux = y$, where L and U are from the factorization for the original sparse matrix equation $Ax = b$

The block triangular solvers are shown in Figure 4.1 and Figure 4.2. The computation flows of the block algorithms are the same as those of element-wise algorithms. In Figure 4.1 and Figure 4.2, B_i denote $b_{S(i):S(i+1)-1}$. Again in the block triangular solves, we assume that the block structure of $\boldsymbol{L}$ is Stored in a column oriented manner and $\boldsymbol{U}$ is stored in a row oriented manner.

```
(01) for k = 1 to N
(02)        b = P_k b;
(03)        Solve L_kk B_k = B_k;
(04)        for i = k + 1 to N with L_ik ≠ 0
(05)              B_i = B_i - L_ik·B_k;
(06)        endfor
(07) endfor
```

Figure 4.1: Block forward substitution to solve $\mathbf{P_1 L_1 P_2 L_2} \cdots \mathbf{P_N L_N} y = b$. y is stored in b at output. B_i denote $b_{S(i):S(i+1)-1}$.

```
(01) for k = N to 1
(02)        Solve U_kk B_k = B_k;
(03)        for j = k − 1 to 1 with U_kj ≠ 0
(04)              B_j = B_j - U_jk·B_j;
(05)        endfor
(06) endfor
```

Figure 4.2: Block back substitution to solve $Ux = y$. y is stored in b at input, and x is stored in b at output. B_i denote $b_{S(i):S(i+1)-1}$.

Recall that in the sparse LU factorization with 2-D data mapping, a column block is distributed onto a column of processors and a row block is distributed onto a row of processors. So no matter whether row oriented or column oriented, the computation of each stage can be overlapped. The column oriented approach requires less communication, simple runtime control, less memory and integer operation overhead, and better data locality, but multiple steps can not be overlapped significantly.

4.2 Parallel Triangular Solve

In this section, we implement the parallel version of triangle solve. Triangular solving has two phases, the first phase is to solve $Ly = b$ and the second phase is to solve $Ux = y$ where y is the temporary result from the first phase. To save space, we will use the same memory location for b, y and x, which means the result of y will cover b while the result of x will cover y, and x is the solution of $Ax = b$.

We still use the data mapping scheme as in Chapter 3, therefore we use the resulting L and U blocks obtained from LU factorization in the same processor.

At first, we divide the column b into several blocks B_i where $1 \leq i \leq N$ and B_i denote $b_{S(i):S(i+1)-1}$. Then we distribute these B blocks into the processors which own the first block column of A according to the rule that the processor which own L_{k1} will own block B_k, therefore the initial b values are distributed on the first column blocks. Then as shown in Figure 4.3, the first column will begin the first step of $Ly = b$ and update the corresponding b blocks. Once it's done, the first column processors will send the updated B blocks it own to the second column processors which will update B blocks in the second step of $Ly = b$. Keeping in this way, the processors which own the last column of A blocks will have the final results of $Ly = b$ stored in its own B blocks.

Next, the second phase of triangle solving, $Ux = y$ begins at the processors

which owns the last column of A blocks. Each processor here will update its own B blocks as in $Ly = b$, once it's done, it will send its own B blocks to its left neighbor, and next its left neighbor will update B blocks in the same way. Keeping this way for N steps, the processors which own the first column of A blocks will get the final result of $Ax = b$ and store it in its own B blocks.

Finally, we collect all B blocks from the processors which own the first column of A blocks and put it back in b, which is the final result.

The two phases of parallel block triangular solves are shown in Figure 4.3 and Figure 4.4. In Figure 4.3 and Figure 4.4, B_i denote $b_{S(i):S(i+1)-1}$.

4.3 Performance of Parallel Triangle Solves

In this section, we show the time performance of the parallel triangle solver designed in Section 4.2.

Figure 4.1 shows the time comparison of LU factorization and triangular Solve of MPI code on Cray T3E using 8 processors. We can see the time needed for the triangle solve phase is less than 35% of the time needed for LU factorization for most test matrices. Therefore the asynchronous computation scheduling which can be used to utilize more parallelism is unnecessary here, since it will involve more difficulty in the design algorithm but the time improvement is ignored compared with time used for LU factorization.

The time performance for LU factorization and triangle solve is compared by the same MPI code, therefore this comparison is acceptable.

We didn't implement the parallel triangle solve using SHMEM routines since SHMEM routines can only be used on Cray T3E machines, thus many users who have access to Origin or Meico can't use it. The SHMEM code is only implemented for LU factorization in Chapter 3, since we want to find the highest GFLOPs using any language and any platform. We already see we can reach 10GFlops under Cray T3E, which is the highest in the world.

```
(01)    V = P_1 P_2 ··· P_N ;
(02)    b = V^{-1} b;
(03)    for k = 1 to N
(04)      if the processor owns column k
(05)        if L_kk is local
(06)          solve L_kk B_k = B_k
(07)          send B_k to all the processors in the same column
(08)        else
(09)          receive B_k from the processor which owns L_kk
(10)        endif
(11)        update B_j = B_j − L_jk B_k if L_jk is local and L_jk ≠ 0
(12)        send the updated B blocks it owns to its right neighbor
(13)      endif
(14)      if the processor owns column k + 1
(15)        receive the updated B blocks from its left neighbor
(16)      endif
(17)    endfor
```

Figure 4.3: Forward substitution to solve $\mathbf{P_1 L_1} \cdots \mathbf{P_N L_N} y = b$. y is stored in b for output. B_i denote $b_{S(k):S(k+1)-1}$, and $bsize(k)$ denote $S(k+1) - S(k)$.

```
(01)      for k = N to 1
(02)        if the processor owns column k
(03)          if U_kk is local
(04)            solve U_kk B_k = B_k
(05)            send B_k to all the processors in the same column
(06)          else
(07)            receive B_k from the processor which owns U_kk
(08)          endif
(09)          update B_j = B_j - U_jk B_k if L_jk is local and L_jk ≠ 0
(10)          send the updated B blocks it owns to its left neighbor
(11)        endif
(12)        if the processor owns column k - 1
(13)          receive the updated b blocks from its right neighbor
(14)        endif
(15)      endfor
```

Figure 4.4: Back substitution to solve $Ux = y$. y is stored in b at input, and x is stored in b at output. B_i denote $b_{S(k):S(k+1)-1}$, and $bsize(k)$ denote $S(k+1)-S(k)$.

Table 4.1: Comparison of LU factorization and Triangular Solve of MPI code with 8 processors. Time is in seconds.

Matrix	Time of Triangular Solving	Time of LU Factorization
OLAF1	2.42	11.38
af23560	4.91	18.95
vavsis	33.81	94.70
e40r0100	3.09	8.39
ex11	3.49	35.21
goodwin	0.71	2.55
jpwh991	0.11	0.28
memplus	6.01	14.17
orsreg1	0.27	0.69
raefsky4	4.67	58.40
saylr4	0.54	1.16
sherman3	1.80	1.12
sherman5	0.92	0.60
TIa	0.84	1.57
TIb	10.44	20.91
TId	1.56	2.92
wang3	17.54	119.70

Chapter 5

Implementation of MPI S^+ software and Comparison with SHMEM

In Chapter 3, the parallel programs used to verify the effectiveness of our new space allocation methods are implemented by SHMEM routines, which are specific to Cray T3E and T3D platform.

The SHMEM routines are data passing library routines similar to message passing library routines. They can be used as an alternative to message passing routines such as Message Passing Interface (MPI) or Parallel Virtual Machine (PVM). Like the message passing routines, the SHMEM routines pass data between cooperating parallel processes. SHMEM routines support remote data transfer throughput operations, which transfer data to a different PE, and get operations, which transfer data from a different PE.

SHMEM is much faster than MPI in that one processor can just write or get data from the buffer area of another processor without the awareness of the second one by SHMEM code. However, it can only be implemented on Cray machines and therefore a program written by SHMEM routines can not be ported

to other platforms such as ORIGIN 2000, or Meico machines. Thus the parallel code we implemented in Chapter 3 has only theoretical importance to verify the effectiveness of our new space saving methods. In order to use this sparse matrix solver software called S^+ on different platforms for different users, we must implement its equivalent Message Passing Interface (MPI) version, since MPI is a widely used language on almost all parallel systems and its parallel performance is robust and stable.

Since SHMEM is much faster than MPI in communication between different processors on Cray by utilizing Cray T3E communication properties, it is not surprising to see that the new MPI version of S^+ should be slower than SHMEM version. In Section 5.1, we will show how we implemented the MPI version of S^+, and in Section 5.2, we will compare the time performance of these two different codes on Cray T3E.

5.1 Implementation of S^+ MPI code

Since we solve the matrix equation on parallel machines, each processor owns one part of the L and U structure. It finishes its computation during each step of Gaussian elimination, (symbolic factorization, LU factorization and triangle solve) by communicating with other processors to exchange data blocks frequently. Receiving and sending of data blocks between different processors can be implemented efficiently by SHMEM routines or MPI routines.

Before we present the MPI implementation of S^+, we first recall the implementation of SHMEM code. SHMEM is one kind of parallel routine which can write or get data from the buffer area of another processor without the awareness of the second one. Therefore, when one processor needs to send some kind of data to another processor, it just writes the data to the destination buffer and then sets one special bit in this buffer to 1. When the second processor needs to use this data, it first checks that bit. If the bit is set to 1, it knows the data has

already arrived and it just transfers the data in that buffer area to its actual position in memory. Certainly, different data should correspond to different buffer areas on each processor, and all the processors know the situation in advance.

The advantage of the SHMEM code in communication is that each processor can just read or write data from another processor's buffer area directly. At this time, the second processor may do something else, but it doesn't matter. The receiving processor only needs to check the checking bit to see if the data is received when it needs that data.

However, MPI routines have no such advantages in communication. They must use MPI-Recv or MPI-Send functions to perform the above communication.

Now, let us explain some communication functions in S^+ MPI code and its functionality.

There are four types of communication that need buffering:

1. Pivoting along a processor column, which includes communicating pivot positions and multicasting pivot rows. We call the buffer for this purpose *Pbuffer*.

2. Multicasting along a processor row. The communicated data includes L_{kk}, local nonzero blocks in $L_{k+1:N,\ k}$, and pivoting sequences. We call the buffer for this purpose *Cbuffer*.

3. Row interchange within a processor column. We call this buffer *Ibuffer*.

4. Multicasting along a processor column. The data includes local nonzero blocks of a row panel. We call the buffer *Rbuffer*.

Corresponding to the above different communication buffers, we provide the following functions between different processors:

1. SP-pivot-send and SP-pivot-get. The processor which owns block A_{kk} will send the pivot at step k to each processor on the same row.

2. SP-col-send and SP-col-get. Each processor which owns column k will send its own blocks of block[k+1,...,n][k] to the processors which have the same row number as that processor.

3. SP-row-send and SP-row-get. Each processor which owns row k will send its own blocks of block[k][k+1,...,n] to the processors which have the same column number as that processor.

All the above communication functions are implemented by using MPI-Recv and MPI-Send functions.

5.2 Comparison of MPI with SHMEM

In this section we will show the comparison of time performance of S^+ MPI code and SHMEM code. Since the major part of Gaussian Elimination is LU factorization in time and space requirement, we only consider LU factorization here.

Table 5.1 provides the comparison of LU time performance of SHMEM code and MPI code on Cray T3E with 4, 8, 16 processors. From this table, we can see the MPI code is really slower than the SHMEM code under Cray T3E. With 4 processors, the MPI code is 28% slower than the SHMEM code on average, and with 8 processors, the MPI code is 29% slower than the SHMEM code on average, while with 16 processors, the MPI code is 41% slower than the SHMEM code on average.

Table 5.1: Comparison of parallel performance of MPI and SHMEM code with 4, 8, 16 processors. Time is in seconds. A "-" implies the data is not available due to insufficient memory.

Matrix	4		8		16	
	SHMEM	MPI	SHMEM	MPI	SHMEM	MPI
OLAF1	15.87	19.87	9.27	11.38	6.14	8.52
af23560	26.59	33.45	14.91	18.95	8.96	13.05
vavsis	-	-	82.52	94.70	45.00	56.77
e40r0100	11.32	13.70	6.03	8.39	3.80	6.23
ex11	56.45	65.02	30.63	35.21	16.61	20.53
goodwin	3.10	4.125	1.97	2.55	1.20	2.54
jpwh991	0.21	0.34	0.14	0.28	0.12	0.35
memplus	19.82	24.29	11.51	14.17	6.84	9.68
orsreg1	0.59	0.94	0.41	0.69	0.41	0.81
raefsky4	100.2	109.15	51.74	58.40	28.36	33.64
saylr4	1.02	1.76	0.68	1.16	0.53	1.30
sherman3	0.90	1.39	0.51	1.12	0.41	1.31
sherman5	0.32	0.69	0.24	0.60	0.21	0.68
TIa	1.29	1.96	0.78	1.57	0.51	1.24
TIb	27.59	37.45	15.78	20.91	9.54	13.69
TId	3.17	4.96	2.20	2.92	1.21	2.37
wang3	-	-	103.23	119.70	54.28	66.85

Chapter 6

Conclusions and Future Directions

6.1 Summary of contributions

In this thesis, we implemented a time and space efficient parallel method for sparse matrix solving(i.e., Gaussian elimination) on distributed memory machines.

In the first step of sparse Gaussian elimination, we implemented the matrix ordering algorithm, i.e., minimum degree ordering from J.W.Liu. However, the ordering may introduce a lot of zeros on the diagonal, bring in more burden to the symbolic factorization. Therefore, we implemented a new transversal algorithm to reduce the portion of zeros on the diagonal to less than 1%, while the time requirement for this algorithm is still linear to the number of nonzeros of the matrix. The ordering and transversal is the minor part of Gaussian elimination according to the time requirement but this is very important to later steps.

The second step concerns LU factorization. We proposed two space optimization schemes. These two space optimization techniques used in LU factorization, Delayed space allocation and Space reclamation, effectively reduce memory re-

quirements when static symbolic factorization creates an excessive amount of extra fill-ins. This new algorithm with dynamic space management exhibits competitive sequential space and time performance compared to SuperLU for the tested matrices. The parallel code becomes more robust in handling different classes of sparse matrices.

Finally, we implemented the parallel triangular solving. In this step, we still use the same data mapping scheme, therefore, the data shuffling is avoided. We use forward substitution for the first phase $Ly = b$ and use backward substitution for the second phase $Ux = y$ while x, y and b use the same space to reduce the space requirement. At first, the initial column b is distributed to all processors which owns the first column blocks of the matrix. In the first phase $Ly = b$, with k going from 1 to n, the updated b at each step k will move from left to right,i.e., each processor which owns column blocks k will send its own B blocks to its right neighbor. In the phase of $Ux = y$, with k going from n to 1, the updated b at each step k will move from right to left, i.e., each processor which owns column blocks k will send its own B blocks to its left neighbor. Finally, all the b blocks will be collected together to constitute the final solution.

Considering the advantage of our S^+ method in time and space performance on distributed memory machines , we released the MPI version of S^+ to the public. You can download it from "http://www.cs.ucsb.edu/research/S+". The portability of this software is very good. You can load and use it on Cray T3E, T3D machines, SGI 2000 machine.

6.2 Future research directions

Since the ordering of the matrix before Gaussian Elimination will have an important influence on the performance of Gaussian Elimination, it is an interesting research direction to study impact of the matrix ordering. Until now, there is no optimal method to handle the matrix ordering. Some other approaches that

handle nonsymmetric matrices using the multifrontal method [1] and static pivoting [21] may be valuable to our S^+ method, which needs further investigation.

The second direction is the improvement of our S^+ MPI code which is slower than the SHMEM code due to the communication pattern. We believe we can still improve our MPI code by providing more parallelism in both computation and communication to reduce the time difference between MPI and SHMEM code.

Bibliography

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

[2] Richard L. Burden and J. Douglas Faires. *Numerical Analysis*. PWS Publishing Company, fifth edition, 1993.

[3] T. Davis. User's guide for the Unsymmetric-pattern Multifrontal Package (UMFPACK). Technical Report TR-93-020, Computer and Information Sciences Department, University of Florida, June 1993.

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

[5] T. A. Davis, J.R. Gilbert, E. Ng, and B. Peyton. Approximate Minimum Degree Ordering for Unsymmetric Matrices. Talk presented at XIII Householder Symposium on Numerical Algebra, June 1996. Journal version in preparation.

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

[7] J. Demmel, S. Eisenstat, J. Gilbert, X. S. 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.

[8] 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.

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

[10] I. S. Duff, A. M. Erisman, and J. K. Reid. *Direct Methods for Sparse Matrices*. Clarendon Press, 1986.

[11] C. Fu, X. Jiao, and T. Yang. A Comparison of 1-D and 2-D Data Mapping for Sparse LU Factorization with Partial Pivoting. In *Proc. of Eighth SIAM Conference on Parallel Processing for Scientific Computing*, March 1997.

[12] 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.

[13] C. Fu, X. Jiao, and T. Yang. Parallel Sparse LU Factorization with Partial Pivoting on Distributed Memory Architectures. Tech Rep. TRCS97-11, UCSB Computer Science, 1997.

[14] 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.

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

[16] A. George and E. Ng. Symbolic Factorization for Sparse Gaussian Elimination with Partial Pivoting. *SIAM J. Scientific and Statistical Computing*, 8(6):877–898, November 1987.

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

[18] J. A. George and J. W.-H. Liu. The Evolution of the Minimum Degree Ordering Algorithm. *SIAM Review*, 31:1–19, 1989.

[19] 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.

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

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

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

[23] K. Shen, X. Jiao, and T. Yang. Elimination Forest Guided 2D Sparse LU Factorization. In *Proceedings of the 10th ACM Symposium on Parallel Algorithms and Architectures*, pages 5–15, June 1998. Available at *www.cs.ucsb.edu/research/RAPID.html*.

[24] M. Yannakakis. Computing the Minimum Fill-In is NP-Complete. *SIAM J. Alg. Disc. Meth.*, 2:77–79, 1981.