Parallel Sparse Gaussian Elimination with Partial Pivoting and 2-D Data Mapping

by

Xiangmin Jiao

B.S. (Beijing University) 1995

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 Oscar Ibarra
Professor Ömer Egecioglu

July 1997

The dissertation of Xiangmin Jiao is approved:

Chair Date

Date

Date

University of California at Santa Barbara

July 1997

Parallel Sparse Gaussian Elimination with Partial Pivoting and 2-D Data Mapping

by
Xiangmin Jiao

Abstract

Parallel Sparse Gaussian Elimination with
Partial Pivoting and 2-D Data Mapping

by

Xiangmin Jiao
Master of Science in Computer Science

University of California at Santa Barbara

Professor Tao Yang, Chair

Sparse Gaussian elimination with partial pivoting is a fundamental algorithm for many scientific and engineering applications, but it is still an open problem to develop a time and space efficient algorithm on distributed memory machines. In this thesis, we present an asynchronous algorithm which incorporates static symbolic factorization, nonsymmetric L/U supernode partitioning and supernode amalgamation strategies, 2-D block mapping scheme, and irregular parallelism scheduling. The distinguished features of the algorithm are low memory requirements, simple run-time control, and BLAS-3 computation kernel. We demonstrate that the new algorithm can solve very large sparse linear systems and achieve high absolute performance. We have conducted experimental studies on the Cray-T3D and T3E with a set of large nonsymmetric benchmark matrices from various fields, and shown that the new algorithm has achieved breakthrough megaflops performance for this open problem in the literatures. We also present an efficient implementation for symbolic factorization and fast algorithms for supernode partitioning and supernode amalgamation.

Contents

List of Figures

List of Tables

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, therefore I could pursue research in the right direction. I thank Oscar Ibarra and Ömer Egecioglu for serving on my thesis committee. I thank Cong Fu particularly for closely working with him in the projects. I would like to thank the members of the Parallel System Laboratory of the Computer Science Department, particularly Daniel A. Andresen, Kai Shen, and Huican Zhu, with whom it has been great pleasure to work. I thank Martin Rinard and Pedro Diniz for their valuable comments. I would like to thank Stefan Boeriu and San Diego Supercomputing Center for their helps to access the Cray T3D and T3E. Finally I would like to thank my wife Boyan for her constant support.

I would like to acknowledge support from NSF RIA CCR-9409695, NSF CDA-9529418, the UC MICRO grant with a matching from SUN, NSF CAREER CCR-9702640, and ARPA DABT-63-93-C-0064 through the Rutgers HPCD project.

Chapter 1

Introduction

A time and space efficient parallelization for sparse Gaussian elimination with pivoting is the key to solving large sparse nonsymmetric 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.

It is still an open problem to develop an efficient implementation for sparse Gaussian elimination with partial pivoting on distributed memory machines. The reasons for that are various. One reason is that to maintain numerical stability, pivoting operations interchange rows based on numerical values of the matrix elements during the elimination process, which dynamically change the structures of L and U factors. The adaptive and irregular data structure of sparse LU factorization makes the efficient implementation of this algorithm very hard [**?**, **?**]. Another reason is that an efficient sparse LU algorithm has to take advantage of memory hierarchy of modern computers. However this is difficult even on sequential machines [?]. Third, an efficient parallelization also has to be space efficient. However time efficiency and space efficiency may be two conflicting goals. On distributed memory machines, one has to distribute data structures onto multiprocessors and minimize communication buffer for space efficiency, which may cause larger communication volume and less computation and communication overlapping. Because of these reasons, it is very difficult to achieve high speedup compared to the best sequential Gaussian elimination code and maintain low memory requirements.

In this thesis, we present a new approach which achieves time and space efficiency at the same time. The approach together with the previous 1-D approaches [?] is called S^*. The key idea of S^* is to trade more nonzeros and more computations for static structures and better computation kernel. S^* uses the following key strategies at the same time in parallelizing the sparse LU algorithm:

- Using a static symbolic factorization scheme to eliminate the data structure variation caused by dynamic pivoting [?];

- Identifying dense submatrices from the sparse structure obtained by the symbolic factorization scheme so that efficient dense operation kernels can be used to perform most of the computation.

- Developing scheduling techniques to overlap computation and communication to a maximum degree and reduce memory requirements for solving large problems.

With static structures, the processors can spend more time on computation instead of allocating space for new fills and moving data around, which contributes high runtime overhead in dynamic approaches. Static structure also enables S^* to mostly call BLAS-3 level subroutines, while the dynamic approaches mostly call BLAS-2 or BLAS-1 level subroutines. On most current commodity processors with memory hierarchies, BLAS-3 subroutines usually outperform BLAS-2 subroutines substantially when implementing the same numerical functionality [?, ?]. Note that the preprocessing cost from symbolic factorization and supernode partitioning is negligible, and expensive runtime control is avoided, so the sequential performance of S^* is still time efficient after introducing extra computation. By developing effective scheduling techniques, S^* is able to deliver high speedup compared to the S^* sequential code.

The S^* has been previously implemented using 1-D data mapping and two different scheduling strategies [?, ?]. One Scheduling is compute-ahead scheduling which has simple run-time control and low memory requirements but suffers from poor performance. The other uses RAPID system which provides sophisticated graph scheduling tools and run-time execution support [?, ?], and it delivers excellent performance but requires large memory per node. 2-D data mapping scheme is considered more scalable than 1-D data mapping, because it has higher order of parallelism, better load balancing, and lower order of communication requirements. We have developed an algorithm using 2-D data mapping for sparse LU factorization with compute-ahead scheduling. The 2-D algorithm delivers good performance and has simple runt-time control and low memory requirements. We have shown that this algorithm has potential to solve large sparse matrices and achieve high megaflops numbers.

However there are several difficulties in applying the 2-D block mapping to sparse Gaussian elimination with partial pivoting even after the static structure is adopted. First of all, pivoting operations and row interchanges require frequent and well-synchronized inter-processor communication when the submatrices in the same column block are assigned to different processors. To exploit irregular parallelism, we have to deal with irregular and asynchronous communication. Second, the communication pattern of 2-D blocking mapping is more complicated than 1-D mapping. To solve large matrices we have to carefully design the communication buffers. If the communication buffer requirements are excessive, it may exceed available memory on each processor for large systems and/or on large number processors. If communication buffer is too small, it may not expose enough parallelism.

The rest of the paper is organized as follows. Chapter 2 gives the problem definition. Chapter 3 describes the fundamentals of the static approach including static symbolic factorization and dense structure identification. Chapter 4 describes the parallel implementations of S^* with 1-D data mapping. Chapter 5 describes the new algorithm using 2-D data mapping. Chapter 6 discusses the issues of utilizing the factorization results in parallel triangular solve and Newton's method. Chapter 7 concludes the thesis and points out further research directions for the problem.

Chapter 2

Sparse Gaussian Elimination with Partial Pivoting

In this chapter, we briefly review the algorithm using Gaussian elimination with partial pivoting to solve a large sparse linear system $Ax = b$. Generally speaking, sparse Gaussian elimination consists of three steps: **analyze**, **factorize** and **solve** [**?**, **?**], 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 the chapter is organized as follows. Section 2.1, 2.2 and 2.3 discuss the three steps respectively and emphasize the differences between sparse and dense eliminations. Section 2.4 formally defines block-oriented sparse Gaussian elimination. Some conventional notations used in the remaining thesis are introduced in Section 2.5.

2.1 The analyze phase

The analyze phase is the preprocessing step for sparse Gaussian elimination. The key features of the analyze phase are to find a good pivotal sequence to preserve sparsity by using some ordering algorithms, and prepare data structures for efficient execution of the other phases.

Ordering is an important issue in sparse matrix computation, because different order of equations and variables could have 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 a NP-hard problem [**?**]. Some heuristic ordering algorithms have been extensively discussed in [**?**], and better algorithms are still under investigation [**?**]. The research of ordering algorithm is beyond the scope of this thesis. In the thesis, we always assume that the input linear system has been appropriately ordered. In our experiments, we use the column minimum degree ordering implemented in MATLAB [**?**], and then apply transversal obtained from Duff's algorithm [**?**] to have a zero-free diagonal. These two algorithms both help reduce fill-ins.

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 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 [?, ?, ?, ?] and it is much easier to be parallelized on distributed memory machines.

Another important data structure to be prepared during the analyze phase is the block structure. In high performance computations, block algorithms are used extensively to exploit the memory hierarchy and reduce indirect memory accessing [?, ?]. It has been shown that the block-wise algorithms outperform the element-wise algorithms for the same computational functionality. In static approach of sparse Gaussian elimination, the block structure can be identified after symbolic factorization. This process is called *supernode partitioning*.

The analyze phase has lower time complexity and requires less memory than factorize phase, therefore in this research we do not parallelize this part. In Chapter 3 we will present a sequential implementation for symbolic factorization and supernode partitioning, which is considered time and space efficient.

2.2 The factorize phase

Given a nonsingular matrix A, it can be decomposed into two matrices L and U using sparse LU factorization with partial pivoting as shown in Figure 2.1, where L is a unit lower triangular matrix, and U is an upper triangular matrix. The steps of elimination are controlled by loop k. For the elements manipulated at step k, we use i for row indexing and j for column indexing. This notation will be used for the rest of this thesis. During each step of the elimination process, a row interchange may be needed to maintain numerical stability. The result of the factorization process can be expressed by:

$$A = P_1 L_1 P_2 L_2 \cdots P_{n-1} L_{n-1} U,$$

where P_k is an elementary permutation matrix corresponding to the row interchange at step k, and L_k is a unit lower triangular matrix whose kth column contains the multipliers at step k, and U is an upper triangular matrix. For sparse LU factorization, L is defined as [1]

$$L = (L_1 + L_2 + \cdots + L_{n-1}) - (n-2)I, \tag{1}$$

compared to dense LU factorization in which L is defined as

$$L = PP_1 L_1 P_2 L_2 \cdots P_{n-1} L_{n-1}, \tag{2}$$

where $P = P_{n-1} \cdots P_2 P_1$. From the viewpoint of the algorithms, the difference between these two definitions lies in row interchange as shown in line (04) of Figure 2.1. For the definition of (1), only part of row k and m are interchanged, or more precisely only $A_{k,k:n}$ [2] and $A_{m,k:n}$ are interchanged. However for the definition of (2), the whole row k and row m are interchanged. In this thesis, L is defined as (1) unless explicitly noted, because it reduces dynamic change of L, and minimizes the memory requirements of L in the static approach. Note that it is no longer true that $PA = LU$.

There are several other factorizations that can be used for solving nonsymmetric linear systems. One approach is the unsymmetric-pattern multi-frontal method [**?**, **?**] that uses Markowitz ordering and numerical threshold pivoting. Another approach [**?**] is to restructure a sparse matrix into a bordered block upper triangular form and use a special pivoting technique to preserves the structure and to maintain numerical stability at acceptable levels. QR factorization [**?**] is also an alternative method. In this thesis, we are focusing on parallelization of the sparse LU factorization with partial pivoting.

[1] This definition of L for sparse LU factorization is due to George and Ng [**?**].

[2] Here we follow the convention of MATLAB. $A_{k,k:n}$ means $A_{k,k}, A_{k,k+1}, \cdots .A_{k,n}$.

for $k = 1$ **to** $n - 1$
  Find m such that $|A_{mk}| = \max_{k \leq i \leq n}\{|A_{ik}|\}$;
  if A_{mk} = 0, then A is singular, stop;
  Swap row k with row m *if necessary*;
  for $i = k + 1$ to n with $A_{ik} \neq 0$
    $A_{ik} = A_{ik}/A_{kk}$;
  endfor
  for $j = k + 1$ to n with $A_{kj} \neq 0$
    for $i = k + 1$ to n with $A_{ik} \neq 0$
      $A_{ij} = A_{ij} - A_{ik} \cdot A_{kj}$;
    endfor
  endfor
endfor

Figure 2.1: Sparse *LU* factorization with partial pivoting.

```
for k = 1 to n-1
        b = P_k b;
        for i = k+1 to n with L_ik ≠ 0
                b_i = b_i - L_ik · b_k;
        endfor
endfor
```

Figure 2.2: Forward substitution to solve $P_1 L_1 P_2 L_2 \cdots P_{n-1} L_{n-1} y = b$. y is stored in b at output.

```
for k = n to 1
        for j = k+1 to n with U_kj ≠ 0
                b_k = b_k - U_kj · b_j;
        endfor
        b_k = b_k / U_kk;
endfor
```

Figure 2.3: Back substitution to solve $Ux = y$. y is stored in b at input, and x is stored in b at output.

2.3 The solve phase

After decomposing A into L and U factors, the solution of the linear system $Ax = b$ can be solved by two triangular solvers: $P_1 L_1 P_2 L_2 \cdots P_{n-1} L_{n-1} y = b$ and $Ux = y$. Recall that P_i is an elementary pivoting matrix so that $P_i P_i = I$. The two solvers are sketched in Figure 2.2 and 2.3, in which x and y are actually reusing the storage for b. In the algorithms, we assume that the L structure is stored in column oriented manner, and U structure is stored in row oriented manner.

An alternative approach for forward substitution is to solve it during the factorize phase if the right-hand side vector b is known prior to numerical factorization. A minor change to the algorithm in Figure 2.1 will fulfill this job. In the new code we take A as the augmented matrix $A|b$, and change line (08) to "**for** $j = k+1$ to $n+1$ with $A_{kj} \neq 0$", and apply row interchange in line (04) to b as well. In parallel sparse Gaussian elimination, we prefer this version because it simplifies the code and reduces communication volume. This algorithm can be easily extended to multiple right-hand side systems.

The solve phase requires many fewer floating point operations than the factorize phase, so less efforts have been made to parallelize this part. However because of memory issues and Amdahl's law, triangular solve could also become the bottle-neck of sparse Gaussian elimination. In some iterative methods triangular solve is executed repeatedly, in which the performance of triangular solve is as important as factorization. In Chapter 6, we will discuss the parallelization of triangular solve and demonstrate its use in Newton's method.

2.4 Block sparse Gaussian elimination with partial pivoting

Block algorithms have become increasingly important in high performance computers because they substantially improve caching performance and reduce indirect address accessing. In this section, we extend the definition of sparse Gaussian elimination to a block oriented version, so that efficient factorization and triangular solve algorithms could be designed based on this new definition.

Assume that an $n \times n$ matrix A is partitioned into an $N \times N$ block matrix $\boldsymbol{A}$. Let $S(i)$ be the column subscript of the first column in column block i, and $S(N+1) = n+1$. Recall that in section 2.2 we have defined sparse Gaussian elimination decomposing A into

$$A = P_1 L_1 P_2 L_2 \cdots P_{n-1} L_{n-1} U,$$

and defined L as

$$L = (L_1 + L_2 + \cdots + \ L_{n-1}) - (n-2)I.$$

Let us take $P_n = I$ and $L_n = I$, and then the definition becomes

$$A = P_1 L_1 P_2 L_2 \cdots P_n L_n U,$$

and

$$L = (L_1 + L_2 + \cdots + \ L_n) - (n-1)I.$$

Now we can easily extend the definition to block-oriented sparse Gaussian elimination as decomposing $\boldsymbol{A}$ into

$$\boldsymbol{A} = \boldsymbol{P_1 L_1 P_2 L_2 \cdots P_N L_N U},$$

where $\boldsymbol{P_i} = P_{S(i)} \cdots P_{S(i+1)-1}$, $\boldsymbol{L_i} = \boldsymbol{P_i}^{\boldsymbol{-1}} P_{S(i)} L_{S(i)} \cdots P_{S(i+1)-1} L_{S(i+1)-1}$, $\boldsymbol{U} = U$, and

$$\boldsymbol{L} = \boldsymbol{(L_1 + L_2 + \cdots L_N)} - (N-1)I.$$

According to this definition, we can design the algorithm for the block sparse LU factorization as shown in Figure 2.4. In the algorithm, lines (02) to (06) are to find pivoting positions for each column in the column block k. Note that in line (04) all columns in column blocks k to N are interchanged. The other part of the block algorithm is the same as the element-wise algorithm shown in Figure 2.1.

The block triangular solvers are shown in Figure 2.5 and Figure 2.6. The computation flows of the block algorithms are the same as those of element-wise algorithms. In Figure 2.5 and Figure 2.6, 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.

for $k = 1$ **to** N
  for $m = S(k)$ `to` $S(k+1) - 1$
    `Find the pivoting row` t `in column` m;
    `Swap row` t `and row` m `of the columns` $S(k)$ `to` n;
    if A_{mm} `= 0, then` A `is singular, stop;`
    `Scale column` m `and update columns` $m + 1$ `to` $S(k+1) - 1$;
  endfor
  for $i = k+1$ `to` N `with` $\boldsymbol{A_{ik}} \neq 0$
    `Solve` $\boldsymbol{L_{kk} A_{ik} = A_{ik}}$;
  endfor
  for $j = k+1$ `to` N `with` $\boldsymbol{A_{kj}} \neq 0$
    for $i = k+1$ `to` N `with` $\boldsymbol{A_{ik}} \neq 0$
      $\boldsymbol{A_{ij} = A_{ij} - A_{ik} \cdot A_{kj}}$;
    endfor
  endfor
endfor

Figure 2.4: Block sparse LU factorization with partial pivoting.

```
for k = 1 to N
    b = P_k b;
    Solve L_kk B_k = B_k;
    for i = k+1 to N with L_ik ≠ 0
        B_i = B_i - L_ik · B_k;
    endfor
endfor
```

Figure 2.5: 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 $b_{S(i):S(i+1)-1}$.

```
for k = N to 1
    for j = k+1 to N with U_kj ≠ 0
        B_k = B_k - U_kj · B_j;
    endfor
    Solve U_kk B_k = B_k;
endfor
```

Figure 2.6: 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 $b_{S(i):S(i+1)-1}$.

Note that unlike dense Gaussian elimination, it is no longer true for sparse Gaussian elimination that $\boldsymbol{L}$=L. In Chapter 6 we will further discuss the relationship between the definitions of L for sparse and dense Gaussian eliminations.

2.5 Conventional notations

The following notations will be used through the rest of the thesis.

- The *L/U supernode partitioning* divides an $n \times n$ matrix A into an $N \times N$ block matrix $\boldsymbol{A}$. The lower triangular factor of A ($\boldsymbol{A}$) is L ($\boldsymbol{L}$) and the upper triangular factor is U ($\boldsymbol{U}$). We will simply use A (L, U) to denote $\boldsymbol{A}$ ($\boldsymbol{L}$, $\boldsymbol{U}$) when it will not cause confusions. We use $|A|$ ($|L|$, $|U|$) to denote the number of nonzeros and fill-ins in A (L, U). For simplicity we use F to denote $L + U - I$.

- Define $S(i)$ as the global column (row) index of the starting column (row) of the i-th column (row) block. For convenience, we define $S(N+1) = n+1$.

- We use U_{ij} (L_{ij}) to denote the element with coordinate (i, j) in U (L). If an element is not necessary in L or U, we denote it with A_{ij} or a_{ij}. $A_{ij} \neq 0$ means that the element is a nonzero entry or a fill-in. When denoting multiple elements, we follow the convention used in MATLAB. For example with $A_{i,j:j'}$ we mean $A_{ij}, A_{i,j+1}, \cdots, A_{ij'}$. These notations are also applicable to block matrix correspondingly.

- We use U_i to denote row i in U, and L_j to denote column j in L. We define the *structure* of U_i as

  $$U_i = \{j | j \geq i \text{ and } U_{ij} \neq 0\},$$

  and the *structure* of L_j as

  $$L_j = \{i | i \geq j \text{ and } L_{ij} \neq 0\}.$$

  $|U_i|$ ($|L_j|$) denote the number of nonzeros and fill-ins in U_i (L_j). For simplicity, when we mention (the structure of) U_i is *contained* in (the structure of) U_k for $k > i$, we mean $U_i - \{i, i+1, \cdots, k-1\} \subseteq U_k$, and the (the structure of) L_j being *contained* in (the structure of) L_k for $k > j$, means $L_j - \{j, j+1, \cdots, k-1\} \subseteq L_k$.

- A *subcolumn* (*subrow*) is a column (row) in a block. For simplicity, we use a global column (row) index to denote a subcolumn (subrow) in a block.

For example, by subcolumn k in the block $\boldsymbol{U_{ij}}$, we mean the subcolumn in this block with the global column index k where $S(j) \leq k < S(j+1)$. A *compound structure* in $\boldsymbol{A}$ is a block, a subcolumn, or a subrow.

- A compound structure is *nonzero* if it contains at least one nonzero element or fill-in. We use $\boldsymbol{A_{ij}} \neq 0$ to indicate that the block $\boldsymbol{A_{ij}}$ is nonzero. Notice that computations only operate on nonzero compound structures. A compound structure is *structurally dense* if all of its elements are nonzeros or fill-ins.

Chapter 3

Fundamentals of Static Sparse Gaussian Elimination

In this chapter, we discuss some important data structures for sparse matrix computations, present the algorithms of symbolic factorization and supernode partitioning, introduce the definition of elimination forest for sparse Gaussian elimination and review some concepts used through the remainder of the thesis. Our purpose is not only to review previous work, but also to introduce a new compression scheme, an efficient implementation for symbolic factorization, and fast algorithms for supernode partitioning. In Section 3.1 we discuss the conventional data structure for sparse matrix structure and propose a compression scheme for it. Section 3.2 presents symbolic factorization for sparse Gaussian elimination, and demonstrates how the new compression scheme can substantially reduce not only memory requirements but execution times of the algorithm as well. Section 3.3 and 3.4 introduce the techniques of supernode partitioning and supernode amalgamation. Section 3.5 demonstrates how to construct the block structure efficiently after supernode partitioning and amalgamation. Section 3.6 shows the experimental evaluation environments. Section 3.7 demonstrates the impact of the static symbolic factorization on storage, computation requirements and sequential performance. Section 3.8 concludes the chapter.

3.1 Sparse matrix structure

In high-performance sparse matrix code, the nonzero pattern and the numerical values of the matrix are stored separately. The nonzero pattern of a sparse matrix is referred to as *sparse matrix structure* or *sparsity structure* [**?**]. Sparse matrix structure is a very important data structure in sparse matrix computation because of the following reasons.

1. It may have very high memory requirements for large sparse matrices. In some systems, the matrix structure requires comparable space as the numerical values. Inefficient sparse matrix structure limits the solvable problem sizes.

2. Efficient accessing scheme of the matrix structure is important to the overall performance of sparse matrix codes. Sparse matrix structure is *frequently* accessed to find out the address of the numerical values. Inefficient access of matrix structure would slow down the overall performance of sparse matrix codes.

3. The matrix structure is dynamic in many sparse matrix computations. Dynamic changes of the structure may introduce high overhead due to memory management.

These issues make an efficient matrix structure very important for high performance sparse matrix computations. In this section we present a matrix structure which is efficient in terms of space usage as well as accessing time, and it is useful for many large-scale sparse matrix computations.

3.1.1 The conventional matrix structure

The matrix structure is conventionally stored in two parts, say $INDEX$ and $START$. $INDEX$ stores the coordinates of nonzeros, which may be sorted with column coordinates or row coordinates. If it is column (row) major sorted, we call it column (row) oriented structure. We will take column oriented structure as example in the following discussions. Because the nonzeros in the same columns are stored together, so it is redundant to store the column coordinate for each nonzero. In practice $INDEX$ only stores the row coordinates of the nonzeros, and the column coordinates can be derived from $START$, which stores the address of first node of each column in $INDEX$. So *logically* each node of $INDEX$ is a pair $(row, next)$, [1] where row is the row subscript, and $next$ is the address of the next node, and $START$ is an n-vector pointer, where n is the order of the matrix. Figure 3.1 shows a sample matrix and its matrix structure is shown in Figure 3.2. In the conventional matrix structure, the space complexity is $O(|A|)$ where $|A|$ is the number of nonzeros in a sparse matrix A. Readers can refer to [?, ?] for detailed discussion of this storage scheme.

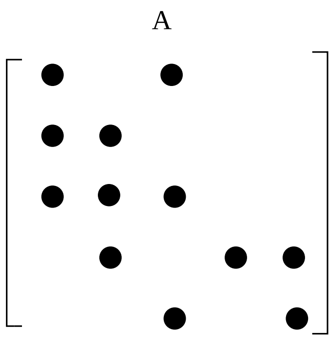

Figure 3.1: A sample sparse matrix.

[1]We call it *logically* because $next$ may not be explicitly stored. For example when $INDEX$ is stored with array, the content of $next$ is implied by the node's own address.

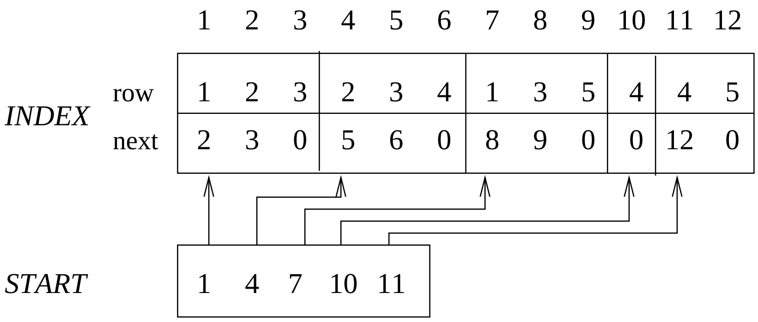

Figure 3.2: Conventional sparse matrix structure.

3.1.2 A general compression scheme for matrix structure

The conventional representation for the sparse matrix structure is not considered efficient to accommodate a very large sparse matrix, especially during the analyze phase of sparse Gaussian elimination when the sparse structure is dynamic. Here we propose a compression scheme which can be easily applied to $INDEX$, and it will potentially save space and time for general large-scale sparse matrix computations. The basic idea is to represent a set of consecutive nonzeros with a single node rather than represent one nonzero with a node. Here by consecutive nonzeros we mean a set of contiguous nonzeros within the same column (in column oriented structure) *or* the same row (in row oriented structure). Again we will take column oriented structure as example. So conceptually a node of $INDEX$ becomes a triple $(first, last, next)$, where $first$ and $last$ specify the smallest and the largest row subscripts in this set of consecutive nonzeros, and $next$ specifies the address of the next node. The structure of $START$ is unchanged. The compressed structure of the sample matrix is shown in Figure 3.3.

We define *consecutivity* c of an $n \times n$ sparse matrix as the average of the number of nonzeros in each consecutive nonzero set in the sparse matrix. Obviously $1 \leq c \leq n$. The compressed sparse matrix structure has space complexity of $O(|A|/c)$. The effectiveness of this compression scheme is determined by c and the data type

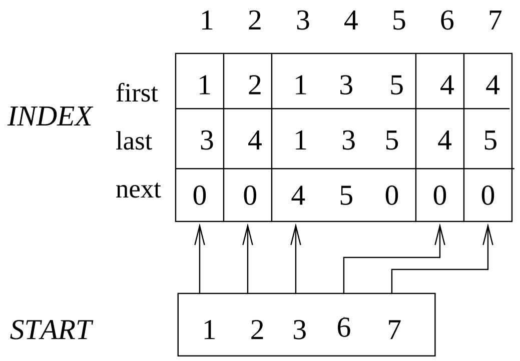

Figure 3.3: Compressed sparse matrix structure.

of subscript and pointer. In static structure when $INDEX$ is stored with array, if $c > 2$, the compressed structure will save space over the conventional structure. In dynamic structure when $INDEX$ is stored with linked list, if the data type of subscript (word) has the same bit length as pointer (word), the compressed structure will save space when $c > 1.5$; if subscript (short word) has half bit length of pointer (word), the compressed structure could save space when $c > 1$ because most architectures align the memory by word, i.e., the operating system has to allocate a two-word space for each node of the conventional structure, so the compressed structure can actually exploit the wasted half-word space. Fortunately for almost all matrices from real world, $c > 2$. [2] We observed that this compression scheme is especially effective for larger and denser sparse matrices for which the memory issue is more critical. For example for an $n \times n$ dense matrices, this scheme only requires $O(n)$ space, while the conventional scheme requires $O(n^2)$ space.

This compression scheme can not only reduce memory requirements, but improve runtime performance. The reason is that it can reduce number of memory accesses for locating a nonzero entry, provide better data locality, and minimize times of memory allocation. This scheme has successfully improved the sym-

[2]We tested a set of benchmark matrices from various sources and all the tested matrices satisfy that $c \gg 2$.

bolic factorization for Cholesky factorization and sparse Gaussian elimination with partial pivoting, and it can also be used in other general sparse matrix computations. In Section 3.2 we will show how this scheme improves symbolic factorization considerably.

3.1.3 Another compression scheme

The compression scheme discussed in subsection 3.1.2 considers the consecutive nonzeros within a column *or* a row, but it does not exploit the structure properties among columns or rows. However, it is very common that part of two columns or rows are identical. For instance in the sample matrix of Figure 3.1, the fourth column is a subset of the fifth column; and the last element (a part) of the fifth column, is also a subset of the third column. Taking advantage of part of these properties should also reduce memory requirements of the matrix structure. This kind of compression scheme is called *subscript compression* in [?]. To avoid confusion, we call the compression scheme we proposed in subsection 3.1.2 *level-1 compression*, because it employs one dimensional structure properties, and call the *subscript compression* [?] *level-2 compression*, because it uses two dimensional structure properties. *Level-2 compression* has different implementations. One simple approach is to check whether the "tails" of two columns (rows) are identical. If they are, the linked lists of these "tails" can be merged. Figure 3.4 shows the matrix structure using level-2 compression of the sample matrix in Figure 3.1.

The criticisms for level-2 compression are that it may slow down the computation because it requires additional indexing to retrieve the nonzeros [?], and it is not applicable to general sparse matrix computations because it is expensive to find out the subset relationship among columns (rows) in general case. However, in some applications level-2 compression can also save space and time if the subset relationship can be known inexpensively. In such cases level-1 and level-2

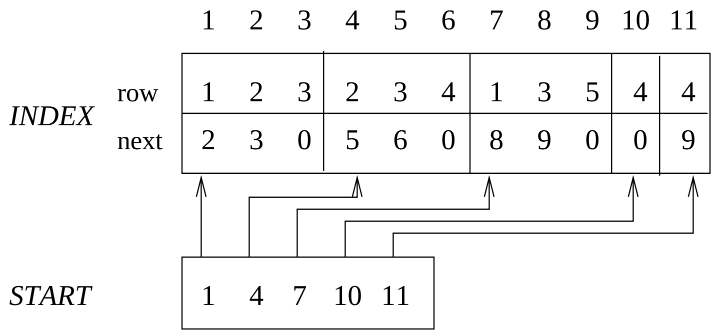

Figure 3.4: The sparse matrix structure using level-2 compression. It makes column 3 and column 5 reuse the ninth element in $INDEX$.

compressions can be used separately or together to improve the implementation. In Section 3.2 we show that employing level-1 and level-2 compressions together in symbolic factorization can accumulate the benefits of both schemes.

3.2 Static symbolic factorization

As we have mentioned in Section 2.1, in our parallelization we use the strategy of static data structure prediction in [?] which avoids dynamic symbolic factorization, identifies the maximal data dependence patterns and minimizes dynamic control overhead. The basic idea is to statically consider all the possible pivoting choices at each step and allocate space for all the possible nonzeros that would be introduced by *any* pivoting sequence that could occur during the numerical factorization. The static approach avoids data structure expansions during the numerical factorization. The symbolic structure can be reused for matrices with the same initial nonzero structure. This is especially useful for iterative methods such as Newton's method in which the Jacobian matrix has the same initial sparsity pattern but with different numerical values. In this section we present the algorithms and an efficient implementation for the symbolic factorization. Note that in the following discussions, we always assume that the input sparse matrix has been ordered appropriately and has no zero diagonal entries.

3.2.1 A symbolic factorization algorithm

Symbolic factorization determines the *upper bound* of the structures of the L and U factors without actually computing the factors numerically. The key idea of the symbolic factorization for sparse Gaussian elimination with partial pivoting 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 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 except 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 3.5.

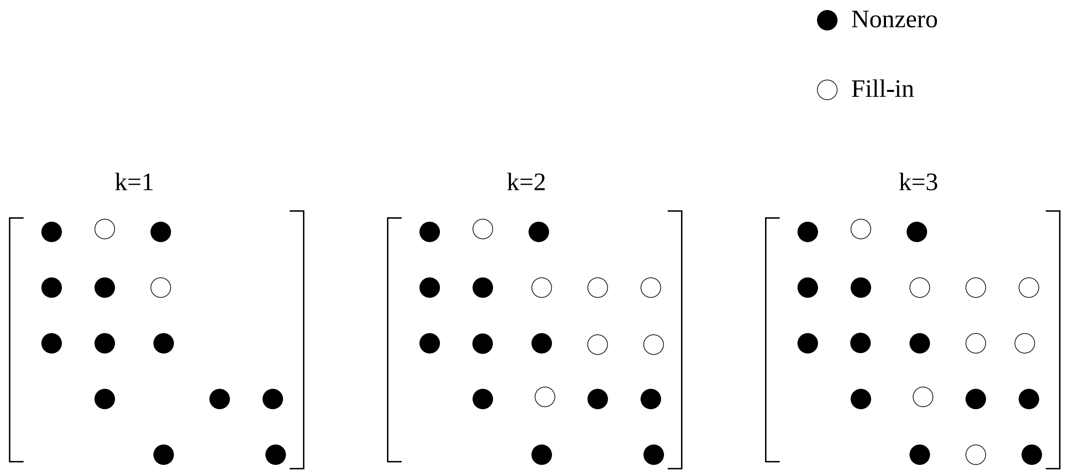

Figure 3.5: The first 3 steps of the symbolic factorization on a sample 5×5 sparse matrix. The structure remains unchanged at step 4.

There are also some other alternative algorithms to compute the symbolic factorization for sparse Gaussian elimination. One is to compute the Cholesky factor of A^TA. This algorithm has been implemented in MATLAB without explicitly computing A^TA [?]. However the algorithm presented here generates a tighter bound than the other methods [?].

This algorithm involves many redundant union operations and hence the time complexity is high. In Section 3.2.3 we will present an efficient algorithm proposed by George and Ng [?], which uses left-looking method and takes advantage of the properties of the elimination forest for sparse Gaussian elimination to reduce redundant union operations. In the next subsection we will first introduce the definition of elimination forest for sparse Gaussian elimination.

3.2.2 Elimination forest for sparse Gaussian elimination

In Cholesky factorization elimination tree (etree) has been used successfully to improve sequential and parallel performances [?]. However unlike Cholesky factorization the elimination tree (forest) for Gaussian elimination is not well defined. In [?] George and Ng defined elimination forest with respect to U. This definition overestimates the dependences of the sparse Gaussian elimination. In this subsection we give a definition of elimination forest for sparse Gaussian elimination with partial pivoting based on static symbolic factorization. With this definition we can identify many useful properties of sparse Gaussian elimination with partial pivoting, and design efficient algorithms for supernode partitioning.

Definition 1 Elimination forest *for Gaussian elimination with partial pivoting is defined as follows. The forest has n nodes, and each corresponds to a row or a column in the matrix A. For* $k < j$*, there is an edge from j to k in the forest if and only if* U_{kj} *is the first off-diagonal nonzero in* U_k *and* $|L_k| > 1$*. j is called the parent of k, and k is the son of j. Column (row) j is the parent column (row) of k, and column (row) k is a child column (row) of j.*

From the viewpoint of the numerical factorization, if j is the parent of k, column j is the first column (except k itself) to be modified at step k. In other words at step k, a column m $(k < m < j)$ *must not* be modified by column k, and column j *must* be modified if we do not distinguish structural nonzero and numerical nonzero at runtime. In George and Ng's definition, they define j is the parent of k if U_{kj} is the first off-diagonal nonzero of U_k. However if $|L_k| = 1$ column j is not modified by column k at step k, so their definition overestimates the dependencies. More importantly, with the new definition we can prove the following properties and use them to guide the supernode partitioning. Based on our definition, the *elimination forest* has following properties.

Lemma 1 *If j is the parent of k in the elimination forest, $L_k - \{k, k+1, \cdots, j-1\} \subseteq L_j$ and $U_k - \{k, k+1, \cdots, j-1\} \subseteq U_j$.*

We outline the proof as follows. Let us use A^k, L^k and U^k to denote the matrix structure of A, L, and U at the end of step k of symbolic factorization respectively. If j is the parent of k, $U^k_{kj} \neq 0$, and L^{k-1}_k must contain an off-diagonal element L^{k-1}_{ik}, and $i \geq j$. So row i is a candidate pivot row of row k. Because at step k *all* the pivot candidate rows are changed to the union of the candidate rows, $L_k - \{k, k+1, \cdots, j-1\} = L^k_k - \{k, k+1, \cdots, j-1\} \subseteq L^k_j \subseteq L_j$, and $U_k = U^k_k \subseteq R^k_i$, where $R^k_i = \{j | A^k_{ij} \neq 0\}$. At step j, i is also a candidate pivot position for j, so $R^k_i - \{1, 2, \cdots, j-1\} \subseteq R^j_i - \{1, 2, \cdots, j-1\} = U^j_j = U_j$, and hence $U_k - \{k, k+1, \cdots, j-1\} \subseteq U_j$.

The following theorem is a straightforward result of this lemma.

Theorem 1 *If there is a path from j to k in the elimination forest, $L_k - \{k, k+1, \cdots, j-1\} \subseteq L_j$ and $U_k - \{k, k+1, \cdots, j-1\} \subseteq U_j$.*

Theorem 1 is the most important property of the elimination forest for sparse Gaussian elimination. In the following discussion of supernode partitioning we will use this theorem to design efficient algorithms. For convenience, in the following discussions by L_k being contained in L_j we mean $L_k - \{k, k+1, \cdots, j-1\} \subseteq L_j$, and by U_k being contained in U_j we mean $U_k - \{k, k+1, \cdots, j-1\} \subseteq U_j$. Another property is summarized as follows.

Theorem 2 *If row j is a candidate pivot row of both row k and row k', and $k < k'$, there is a path from k' to k in the elimination forest.*

This property can be easily verified. We outline the proof as follows. If row j is a candidate pivot row at step k, and the parent of k is k_1, then $k_1 \leq k'$ and $L^k_{j,k_1} \neq 0$, so row j is also the candidate pivot row of k_1. If $k_1 = k'$, we are done. Otherwise following this reasoning, there must be a k_t where $k_t < k'$, and

$$parent(k) = k_1, parent(k_1) = k_2, \cdots, parent(k_{t-1}) = k_t, parent(k_t) = k'.$$

So there is a path from k' to k.

3.2.3 The symbolic factorization algorithm of George and Ng

An efficient algorithm for symbolic factorization was proposed by George and Ng [**?**, **?**]. The algorithm can be described using our definition of *elimination forest*. We outline the key idea of the algorithm as follows.

"Initially $L_j = \{i | i \geq j \text{ and } A_{ij} \neq 0\}$, $R_i = \{j | A_{ij} \neq 0\}$, and $U_i = \phi$ for $1 \leq i \leq n$. At step k, $1 \leq k < n$,

$$L_k = L_k \cup \bigcup L_j \{j | parent(j) = k\} - \{1, \cdots k - 1\},$$

$$U_k = \bigcup U_i \{i | parent(i) = k\} \cup \bigcup R_i \{i | i \text{ is a pivot candidate of } k\} - \{1, \cdots k - 1\},$$

and then set

$$R_i = \phi, i \in \{i | i \text{ is a pivot candidate of } k\}."$$

This algorithm is equivalent to the symbolic factorization presented in Section 3.2.1. It can be easily shown with Theorem 1 and 2 by induction. The time complexity of this algorithm is $O(max(|A|, |U| + |L|)$, which is $O(|F|)$. [**?**] serves as a good source for analysis of this algorithm. The symbolic factorization process of this algorithm for the sample matrix in Figure 3.1 is shown in Figure 3.6.

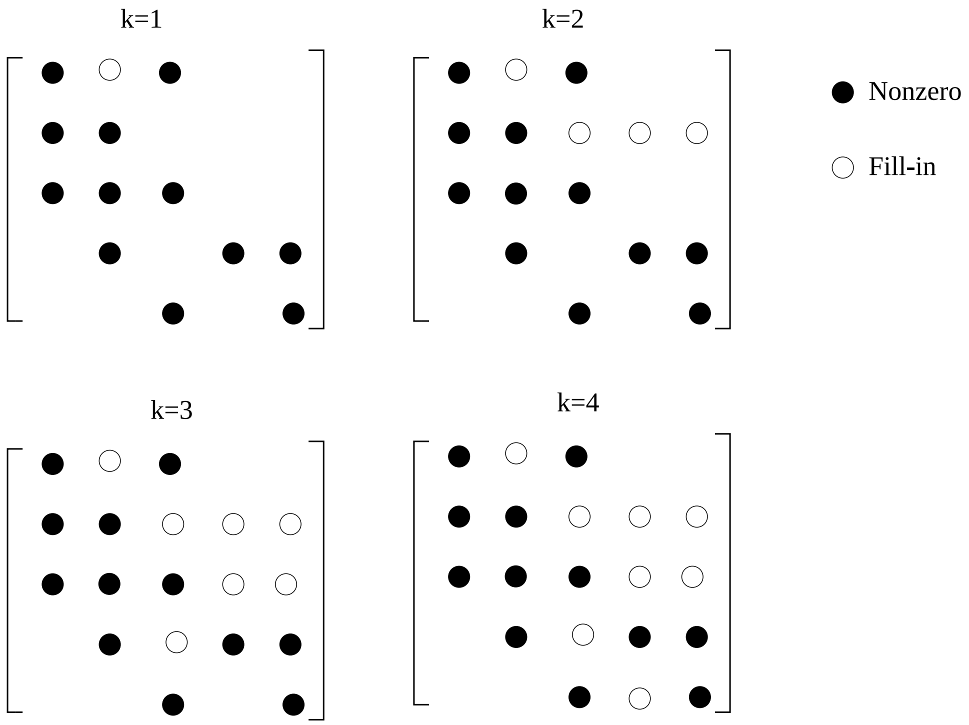

Figure 3.6: The process of the modified symbolic factorization on a sample 5×5 sparse matrix.

3.2.4 An efficient implementation of symbolic factorization

Mathematically the algorithm discussed in Section 3.2.3 has time and space complexity of $O(|U| + |L|)$. However practically by using compressed matrix structure, the complexity might be even lower. Let us first consider the *level-1 compression*. Because consecutive nonzeros are stored with a single node, the space requirements could be lower, and the union operation can be done much faster. Considering two rows R_1 and R_2, when the conventional matrix structure is used, the time complexity of $R_1 \cup R_2$ is $O(|R_1| + |R_2|)$. However when compressed matrix structure is used, the time complexity is only $O(nnode(R_1) + nnode(R_2))$, where $nnode(R_i)$ is the number of nodes in the compressed structure of R_i. So the time and space complexity after using level-1 compression is

$O(max(|L|/c_L + |U|/c_U, |A|/c_A))$, where c_L, c_U, c_A are the *consecutivity* of L, U and A respectively. The space requirement at output is $O(|L|/c_L + |U|/c_U)$.

Level-2 compression can also be used to reduce the space requirements and execution times for the George-Ng symbolic factorization algorithm. The reason is that in the algorithm presented in section 3.2.3, at step k there is very high probability that row k has *only one* child s and $R_i = \phi$ for each candidate pivot row i of row k. In the case, the structure of L_k and U_k are solely determined by the L_s and U_s, which allows us to make L_k (U_k) simply reuse the structure of the L_s (U_s) without any comparison. This scheme was used in [**?**] to improve the implementation. Assume the probability is p, the time and space complexity of this implementation is $O(max((1-p) \cdot |F|, |A|))$, and the space complexity of the output is $O(max((1-p) \cdot |F|, n))$.

Level-1 and level-2 compression schemes can also be used together to improve the algorithm. Now the time and space complexity of the algorithm becomes $O(max((1-p) \cdot (|L|/c_L + |U|/c_U), A/c_A))$ and the space complexity at output is $O(max((1-p) \cdot (|L|/c_L + |U|/c_U), n))$. The experimental results in the subsection 3.2.5 testify our theoretical analysis and show that the last implementation works best in practice.

3.2.5 Impact of compressed structures on symbolic factorization

In this subsection we demonstrate the efficiency of our implementation of symbolic factorization. Our experiments are conducted on Sun Ultra-Sparc 2 dual processor workstation. The clock rate of each processor is 167 MHz. The memory size is 128 Mbytes. The (long) integer is 32 bits long, and short integer is 16 bits long.

Table 3.1 shows the times and memory usages for four different implementations of symbolic factorization. Memory usage in the table is the maximum

Matrix	$\|F\|$	w/o		w/ l-1		w/ l-2		w/ both	
		Time	Space	T	S	T	S	T	S
sherman5	326434	1.03	3.07	0.21	0.86	0.26	1.00	0.10	0.83
lnsp3937	694257	2.17	6.13	0.50	1.61	0.57	1.84	0.21	1.34
lns3937	709337	2.21	6.24	0.50	1.61	0.61	2.00	0.22	1.37
sherman3	625063	1.95	5.51	0.38	1.34	0.29	1.07	0.15	1.06
jpwh991	205016	0.63	1.78	0.13	0.41	0.16	0.49	0.06	0.37
orsreg1	585686	1.80	5.00	0.32	1.02	0.21	0.75	0.10	0.71
saylr4	986203	3.04	8.39	0.57	1.81	0.38	1.30	0.18	1.18
goodwin	3508144	11.2	33.6	1.62	7.57	1.69	8.62	0.53	7.10
e40r0100	9587091	-	-	4.13	16.1	4.56	19.3	1.13	12.7
fidap011	26796090	-	-	9.30	34.2	7.65	26.8	2.38	25.3
raefsky4	36954284	-	-	11.7	41.7	7.72	36.3	2.34	28.6
inaccura	12393720	-	-	4.43	20.8	6.15	29.4	1.47	20.1
af23560	20443661	-	-	6.02	20.6	9.87	31.2	2.47	17.1
vavasis3	53930033	-	-	11.8	67.4	10.7	54.0	3.61	42.9

Table 3.1: Impact of compression schemes on symbolic factorization. Time is in seconds, and space is in Mbytes. For the last matrix the subscript uses long integer, and for all the other matrices the subscript uses short. A "–" means that the test could not complete due to excessive paging.

space for $INDEX$ during the execution of symbolic factorization. It can be seen that with either level-1 or level-2 compression, the code is 3.5 to 5 times as fast as that without compression, and space is only about 1/4 to 1/5.5 of the code without compression. For the code without compression, some tests cause page thrashing and the results are not collected. For about half matrices the code with level-1 compression outperforms the code with level-2 compression, and for the others level-2 compression is better. That is because the two compression schemes depend on the different properties of the matrix structures. However, with both compression schemes the code delivers the best performance and requires least space. This code is about 3 to 4 times faster than the code with either compression, and for the small testing examples, it is about 10 to 20 times faster than the code without compression.

Based on the above experiments,we conclude that the compression scheme we proposed in subsection 3.1 is very effective in symbolic factorization for Gaussian elimination with partial pivoting. The experiments have also verified that the execution time of static symbolic factorization is negligible in static sparse Gaussian elimination. Therefore during this phase of research we do not make effort to parallelize this algorithm, and it is reasonable to exclude this part of time when comparing our algorithm with SuperLU [?], which has to include the time for symbolic factorization.

3.3 Supernode partitioning

Supernode partitioning is a commonly used technique to improve caching performance of sparse code [?]. Excellent performance has been achieved in [?, ?, ?] using supernode partitioning for Cholesky factorization. In this section, we present two different kinds of supernode partitioning strategies for sparse Gaussian elimination, and identify the relationship between the two strategies.

3.3.1 L supernode partitioning

An L *supernode* is defined as a group of consecutive columns in which the corresponding L factor has a dense lower triangular diagonal block and the same nonzero pattern below the diagonal block. Based on this definition, in each column block the L part only contains dense subrows. We call this partitioning approach L supernode partitioning. Here by "subrow" we mean the contiguous part of a row in a supernode. A good analysis for defining unsymmetric supernodes in an L factor is available in [?]. Notice that the supernodes may need to be further broken into smaller ones to fit into cache and to expose more parallelism.

In static approach, the L supernode structure can be identified very efficiently by using elimination forest. The input of L supernode partitioning are two n-vectors. One stores $|L_i|$, and the other stores the elimination forest. The key idea of the algorithm is outlined as follows.

"For $1 \leq i < n$, column i and column $i+1$ are in the same L supernode if and only if $parent(i) = i+1$ and $|L_i| = |L_{i+1}| + 1$."

The correctness of this algorithm can be verified by Theorem 1. If $i+1$ is the parent of i, the structure of $L_i - \{i\}$ is a subset of L_{i+1}. So $|L_i| = |L_{i+1}| + 1$ implies that $L_{i+1} = L_i - \{i\}$. On the other hand, if i and $i+1$ are in the same supernode, according to the definition of the L supernode, $L_{i+1,i} \neq 0$ implies $U_{i,i+1} \neq 0$, so $i+1$ is the parent of i, and $|L_i| = |L_{i+1}| + 1$. The time complexity of this supernode partitioning algorithm is $O(n)$.

After an L supernode partition has been obtained on a sparse matrix F, the same partition can be applied to the rows of the matrix symmetrically, which breaks F into an $N \times N$ block matrix. In [?] Fu and Yang showed the following property.

Theorem 3 *Given a sparse matrix A with a zero-free diagonal, after static symbolic factorization and 2-D L/U supernode partitioning are performed on A, each nonzero submatrix in the U factor of A contains only structurally dense subcolumns.*

The proof of this theorem can be found in [?]. According to this theorem, after 2-D L/U supernode partitioning the diagonal blocks are all dense, and the off-diagonal blocks in L only contain dense subrows and the off-diagonal blocks in U only contain dense subcolumns. Some compression scheme can be applied to store these structures. Figure 3.7 shows a partitioned sparse matrix.

3.3.2 U supernode partitioning

U supernode can be defined similarly as L supernode. We define a *U supernode* as a group of consecutive rows in which the corresponding U factor has a

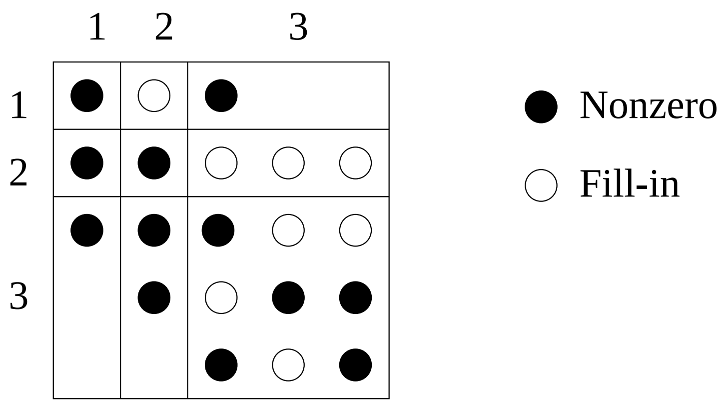

Figure 3.7: An illustration of the L supernode partitioning.

dense upper triangular diagonal block and the same nonzero pattern off-diagonal. The U supernode partitioning is also guided by Theorem 1 because Theorem 1 summarizes a "symmetric" property of L and U factors. The algorithm is outline as follows.

"For $1 \leq i < n$, row i and row $i+1$ are in the same U supernode if and only if $parent(i) = i+1$ and $|U_i| = |U_{i+1}| + 1$."

The correctness of this algorithm can be proved in the exactly same way as that for L supernode partitioning. The time complexity of this supernode partitioning algorithm is $O(n)$.

After U supernode partitioning, the partition can be symmetrically applied to the columns of F. We can easily see that the off-diagonal blocks in U only contain dense subcolumns, but the diagonal blocks may not be dense any more, and it is not necessary that the off-diagonal blocks in L contain only dense subrows. However in the implementation we have to assume that L blocks only contain dense subrows to use dense computation kernel, thus by using U supernode partitioning scheme alone, there could be some new nonzeros introduced. Figure 3.8 shows a partitioned sparse matrix for the sample matrix in Figure 3.5.

L supernode partitioning and U supernode partitioning are tightly related. We can show that the following theorem holds.

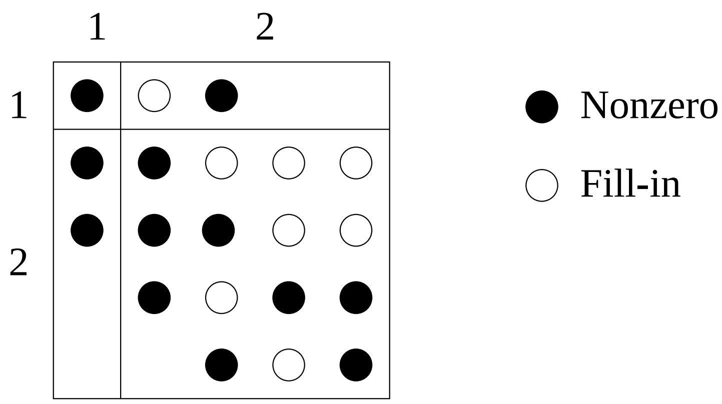

Figure 3.8: An illustration of the U supernode partitioning.

Corollary 1 *Given a matrix F, after L supernode partitioning and U supernode partitioning, if column i and column j are in the same L supernode, then row i and row j must be in the same U supernode.*

This corollary can be easily derived from Theorem 3.

3.4 Supernode amalgamation

For most tested sparse matrices, the average size of a supernode after L/U partitioning is very small. This results in too fine grained tasks. Amalgamating small supernodes can lead to great performance improvement for both parallel and sequential sparse codes because it tends to improve caching performance and reduce interprocessor communication overhead.

There could be many ways to amalgamate supernodes [**?**, **?**]. The basic idea is to relax the restriction that all the columns in an L supernode or the rows in a U supernode must have exactly the same off-diagonal nonzero structure. In Cholesky factorization the amalgamation is usually guided by a supernode elimination tree. The parent could be merged with one of its children if the merging does not introduce too many extra nonzeros into a supernode. If the parent is not consecutive with the child, row and column permutations are needed. The research on amalgamation with permutation for sparse Gaussian elimination with partial pivoting is under investigation. In this section, we present some amalgamation strategies, which involve no permutation.

3.4.1 L supernode amalgamation

L supernode amalgamation amalgamates consecutive L supernodes if their nonzero structures only differ by a small number of entries. The resulting supernode after amalgamation is also referred to as *relaxed supernode* in [**?**]. We can control the maximum allowed differences by an amalgamation factor r. An efficient algorithm for L supernode amalgamation is outlined below.

"For $1 \leq i < n$, column i and column $i+1$ are in the same relaxed supernode if and only if $parent(i) = i+1$ and $|L_{i+1}| < |L_i| + r$."

If $i+1$ is the parent of i, the structure of L_i is contained in L_{i+1}. So $|L_{i+1}| < |L_i| + r$ implies that $|L_{i+1} - (L_i - \{i\})| < r$.

On the other hand, if $i+1$ is not the parent of i, the elements of L_{i+1} and L_i are exclusive. So the structure difference between L_{i+1} and L_i is $|L_{i+1}|+|L_i|$. For large sparse matrices it is reasonable to assume that $|L_{i+1}|+|L_i| > r$ always holds. If $r = 0$, this algorithm shall be L supernode partitioning algorithm described in Section 3.3.1.

This algorithm has time complexity of $O(n)$. After L supernode amalgamation because of Theorem 1, if column i and j are in the same relaxed supernode and $i < j$, $L_i - \{i, i+1, \cdots, j-1\} \subseteq L_j$. So the number of extra nonzeros in L introduced by the amalgamation algorithm is bounded by $r \cdot BSIZE \cdot n/2$, where $BSIZE$ is the maximum size of supernodes.

3.4.2 U supernode amalgamation

We can similarly define the U supernode amalgamation as L supernode amalgamation. We control the maximum difference between consecutive rows in a relaxed supernode by an amalgamation factor r. The algorithm for U supernode amalgamation is outlined as follows.

"For $1 \leq i < n$, row i and row $i+1$ are in the same supernode if and only if $parent(i) = i+1$ and $|U_{i+1}| < |U_i| + r$."

This algorithm has similar properties as L supernode amalgamation. It has time complexity of $O(n)$. The number of extra nonzeros introduced by amalgamation in U is bounded by $r \cdot BSIZE \cdot n/2$.

3.4.3 $L+U$ supernode amalgamation

In the last two subsection, we discussed L and U supernode amalgamation strategies. However neither of the strategies can bound the total extra nonzeros in both L and U factors. In this section, we put the ideas of the two approaches together to bound the total number of extra nonzeros. We define two parameters r_L and r_U to bound the L and U part respectively. The new algorithm is outlined as follows.

"For $1 \leq i < n$, i and $i+1$ are in the same supernode if and only if $parent(i) = i+1$, $|L_{i+1}| < |L_i| + r_L$, and $|U_{i+1}| < |U_i| + r_U$."

This algorithm also has time complexity of $O(n)$. The new nonzeros introduced in L and U factors is bounded by $r_L \cdot BSIZE \cdot n/2 + r_U \cdot BSIZE \cdot n/2$. This algorithm provides us a full control over the extra nonzeros after amalgamation in both L and U.

3.5 Nonzero block identification

After supernode partitioning and applying the partition to both dimensions, A is divided into an $N \times N$ block matrix $\boldsymbol{A}$. The nonzero structures of the block matrix can be determined based on the 2-D L/U partitioning and the matrix structure of F.

For the convenience of implementing numerical factorization, we define a subcolumn in a $\boldsymbol{U}$ block is dense if it has at least one nonzero in the subcolumn, and a subrow in an $\boldsymbol{L}$ block is dense if it has at least one nonzero in the subrow. Based on this definition, we can design an efficient algorithm to identify nonzero compounds in $\boldsymbol{L}$ and $\boldsymbol{U}$. The key idea of the algorithm is outlined as follows:

"Let k be the last column in the column block K. A subrow with global row subscript i in $\boldsymbol{L_K}$ is nonzero if and only if L_{ik} is nonzero. Similarly, if a subcolumn with global column subscript j in $\boldsymbol{U_K}$ is nonzero if and only if U_{kj} is nonzero."

This algorithm is due to Theorem 1. The nonzero structure of each column in an L supernode is contained in the last column of the supernode, so we just need to scan the last column. Similarly, the nonzeros structure of each row in a U supernode is contained in the last row of the supernode, so we just need to scan the last row.

This algorithm has time complexity of $O(N/n|F|)$. An example of partitioned block matrix is shown in Figure 3.9. In our current implementation, an L block is considered a dense block if it has at least one nonzero elements. That allows us maximize the use of BLAS-3 operations, but that may introduce considerable extra nonzeros in L factor, and extra floating-point operations during numerical factorization. That is because the current BLAS subroutines are designed for general dense matrix computations. However as we will show in Section 7.2.3 this problem can be solved by redefining the interface of BLAS to fit for general sparse matrix computation.

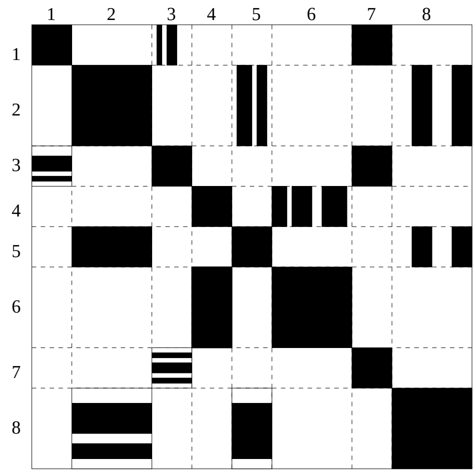

Figure 3.9: An example of partitioned block matrix.

3.6 Evaluation environment

Our experiments were originally conducted on a Cray-T3D distributed memory machine at San Diego Supercomputing Center. Each node of the T3D includes a DEC Alpha EV4(21064) processor with 64 Mbytes of memory. The size of the internal cache is 8 Kbytes per processor. The BLAS-3 matrix-matrix multiplication routine `DGEMM` can achieve 103 MFLOPS, and the BLAS-2 matrix-vector multiplication routine `DGEMV` can reach 85 MFLOPS. These numbers are obtained assuming all the data is in cache and using cache read-ahead optimization on T3D, and the matrix block size is chosen as 25. The communication network of the T3D is a 3-D torus. Cray provides a shared memory access library called `shmem` which can achieve 126 Mbytes/s bandwidth and $2.7\mu s$ communication overhead using `shmem_put()` primitive [?]. We have used `shmem_put()` for the communications in all the implementations.

We have also conducted experiments on a newly acquired Cray-T3E at San Diego Supercomputing Center. Each T3E node has a clock rate of 300 MHz, an 8Kbytes internal cache, 96Kbytes second level cache, and 128 Mbytes main memory. The peak bandwidth between nodes is reported as 500 Mbytes/s and the peak round trip communication latency is about 0.5 to 2 μs [?]. We have observed that when block size is 25, `DGEMM` achieves 388 MFLOPS while `DGEMV` reaches 255 MFLOPS. We have used block size 25 in our experiments since if the block size is too large, the available parallelism will be reduced. In this section we mainly report results on T3E. In some occasions that the absolute performance is concerned, we also list the results on T3D to see how our approach scales when the underline architecture is upgraded. All the results are obtained on T3E unless explicitly stated.

In calculating the MFLOPS achieved by our parallel algorithms, we do not include extra floating-point operations introduced by the overestimation. We use the following formula:

$$\text{Achieved MFLOPS} = \frac{\text{Operation count obtained from SuperLU}}{\text{Parallel time of our algorithm on T3D or T3E}}.$$

The operation count for a matrix is reported by running SuperLU code on a SUN workstation with large memory since SuperLU code cannot run for some large matrices on a single T3D or T3E node due to memory constraint. We also compare the S^* sequential code with SuperLU to make sure that the code using static symbolic factorization is not too slow and will not prevent the parallel version from delivering high megaflops.

3.7 Impact of static symbolic factorization

3.7.1 Impact on storage and computation amount

We have tested the storage impact of overestimation for a set of testing matrices from various sources. The results are listed in Table 3.2. The fourth column in the table is the original number of nonzeros, and the fifth column measures the symmetry of the structure of the original matrix. The bigger the symmetry number is, the more nonsymmetric the original matrix is. A unit symmetry number indicates a matrix is symmetric, but all matrices have nonsymmetric numerical values. We have compared the number of nonzeros obtained by the static approach and the number of nonzeros obtained by SuperLU, as well as that of the Cholesky factor of A^TA, for these matrices. The results in Table 3.2 show that the overestimation usually leads to less than 50% extra nonzeros than SuperLU scheme does. Extra nonzeros do imply additional computational cost. For example, one has to either check if a symbolic nonzero is an actual nonzero during a numerical factorization, or directly perform arithmetic operations which could be unnecessary. If we can aggregate these element-wise floating-point operations and maximize the use of BLAS-3 subroutines, the sequential code performance will still be competitive. Even the fifth column of Table 3.2 shows that the floating-point operations from the overestimating approach can be up to 5 folds, the results in Section 3.7.2 will show that actual ratios of running times are much less.

3.7.2 Impact on sequential performance

In this subsection we demonstrate how the overestimation of nonzeros by the static factorization affects the execution time of numerical factorization. We compare the performance of the S^* sequential code with SuperLU code performance in Table 3.3 and 3.4 for those matrices from Table 3.2 that can be executed on a single T3D or T3E node. We also introduce two other matrices to show how well the method works for larger matrices and denser matrices. One of the two matrices is `b33_5600` which is truncated from `BCSSTK33` because of the current sequential implementation is not able to handle the entire matrix due to memory constraint, and the other one is `dense1000`.

Though the static symbolic factorization introduces a lot of extra computation as shown in Table 3.2, the performance of S^* after 2-D L/U partitioning is consistently competitive to that of highly optimized SuperLU. The absolute

Matrix	Order	$\lvert A\rvert$	$\frac{\lvert A+A^T\rvert}{\lvert A\rvert}$	factor entries Static	SuperLU	A^TA	ratio
sherman5	3312	20793	1.220	326434	250108	424512	3.41
lnsp3937	3937	25407	1.131	694257	453904	933975	5.04
lns3937	3937	25407	1.131	709337	459002	945317	4.74
sherman3	5005	20033	1.000	625063	443349	786061	3.57
jpwh991	991	6027	1.053	205016	141927	256563	3.47
orsreg1	2205	14133	1.000	585686	414699	737669	3.76
saylr4	3564	22316	1.000	986203	670964	1258540	3.85
goodwin	7320	324772	1.358	3508144	3128620	5195434	3.11
e40r0100	17281	553562	1.000	9587091	8171471	14655687	3.11
fidap011	16614	1091362	1.000	26796090	25497683	34064650	1.90
raefsky4	19779	1316789	1.000	36954284	26804225	46981501	2.81
inaccura	16146	1015156	1.000	12393720	9941478	16721768	2.90
fidap011	16614	1091362	1.000	26796090	25497683	34064650	1.90
vavasis3	41092	1683902	1.999	53930033	49192880	65248848	1.43

Table 3.2: Testing matrices and their statistics. The last column specifies the ratio of number of floating-point operations which is defined as $flop(S^*)/flop(SuperLU)$.

single node performance that has been achieved by the S^* approach on both T3D and T3E is consistently in the range of $5-10\%$ of the highest `DGEMM` performance for those matrices of small or medium sizes. Considering the fact that sparse codes usually suffer poor cache reuse, this performance is reasonable. In addition, the amount of computation for the testing matrices in Table 3.3 and 3.4 is small, ranging from 18 to 107 million double precision floating-point operations. Since the characteristic of the S^* approach is to explore more dense structures and utilize BLAS-3 kernels, better performance is expected on larger or denser matrices. This is verified on a matrix `b33_5600`.

For even larger matrices such as `vavasis3`, we cannot run S^* on one node, but as shown later, the 2-D code can achieve 32.8 MFLOPS per node on 16 T3D processors. Notice that the megaflops performance per node for sparse Cholesky

Matrix	S^* Approach		SuperLU		Time Ratio
	Seconds	Mflops	Seconds	Mflops	S^*/SuperLU
sherman5	2.87	8.81	2.39	10.57	1.21
lnsp3937	6.12	6.90	4.94	8.56	1.27
lns3937	6.67	6.71	5.30	8.45	1.26
sherman3	6.06	10.18	4.27	14.46	1.56
jpwh991	2.11	8.24	1.62	10.66	1.34
orsreg1	6.03	10.37	4.09	15.25	1.66
saylr4	10.38	10.31	7.18	14.95	1.60
goodwin	43.72	15.3	-	-	-
b33_5600	110.5	22.9	-	-	-
dense1000	10.48	63.6	19.6	34.0	0.53

Table 3.3: Sequential performance on T3D: S^* versus SuperLU. A "-" implies the data is not available due to insufficient memory.

reported in [?] on 16 T3D nodes is around 40 MFLOPS, which is also a good indication that S^* single-node performance is satisfactory.

3.8 Conclusions and contributions

In this chapter, we proposed a new compression scheme for sparse matrix structure, which is applicable for general large-scale sparse matrix computations. We demonstrated that this compression scheme substantially reduced space requirements and execution times for symbolic factorization. We proposed a new definition of elimination forest for sparse Gaussian elimination with partial pivoting based on symbolic factorization, and summarized some very important properties of the elimination forest. By using the new definition of elimination forest, we defined a U supernode partitioning strategy and identified the relationship between U supernode and L supernode. A new amalgamation algorithm is proposed by integrating the concepts of L and U supernode partitioning, which provides a full control over the extra nonzeros in both L and U factors introduced

Matrix	S^* Approach		SuperLU		Time Ratio
	Seconds	Mflops	Seconds	Mflops	S^*/SuperLU
sherman5	0.94	26.9	0.78	32.4	1.22
lnsp3937	2.0	21.1	1.73	24.4	1.16
lns3937	2.19	20.4	1.84	24.3	1.19
sherman3	2.03	30.4	1.68	36.7	1.21
jpwh991	0.69	25.2	0.56	31.0	1.23
orsreg1	2.04	30.7	1.53	40.9	1.31
saylr4	3.53	30.3	2.69	39.8	1.31
goodwin	17.0	39.4	-	-	-
b33_5600	45.0	56.2	50.0	50.6	0.9
dense1000	4.04	165.0	8.39	79.4	0.48

Table 3.4: Sequential performance on T3E: S^* versus SuperLU. A "-" implies the data is not available due to insufficient memory.

by amalgamation.

Chapter 4

Parallel Sparse LU Factorization with 1-D Data Mapping

In this chapter [1], we discuss the parallel sparse LU factorization with partial pivoting and 1-D data mapping. We will present two different approaches, one of which uses compute-ahead scheduling and block-cyclic mapping, and the other uses RAPID system for scheduling and run-time support. The former approach has simple run-time control and low memory requirements, but suffers from poor load balancing and scheduling. The latter approach achieves excellent performance, but has high memory requirements. The rest of the chapter is organized as follows. Section 4.1 gives the program partitioning and task definition. Section 4.2 presents the data mapping schemes used in 1-D approaches. Section 4.3 outlines the scheduling schemes and run-time support for the methods, and discusses the advantages and disadvantages of both methods. Section 4.4 presents the performance of the both implementations. Section 4.5 concludes the chapter.

[1] Part of this chapter is extracted from Cong Fu's Ph.D. thesis with permission from Fu.

4.1 Program partitioning

After dividing a sparse matrix A into submatrices using the L/U supernode partitioning, we need to partition the LU code accordingly and define coarse grained tasks that manipulate on partitioned dense data structures. In 1-D partitioning scheme, we partition the code following L supernode structures. Typically there are two types of tasks. 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 $U_{kj} \neq 0$.

An outline of the partitioned sparse LU factorization algorithm with partial pivoting is described in Figure 4.1. The code of $Factor(k)$ is summarized in Figure 4.2. It uses BLAS-1 and BLAS-2 subroutines. The computational cost of the numerical factorization is mainly dominated by $Update()$ tasks. The function of task $Update(k, j)$ is presented in Figure 4.3. The lines `(05) and (12)` are using dense matrix multiplications.

Instead of performing the row interchange to the right part of the matrix right after each pivoting search, a technique called "delayed-pivoting" is used [?]. In this technique, 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, i.e., interchange rows. Delayed-pivoting is important, especially to the parallel algorithm, because it is equivalent to aggregating multiple small messages into a larger one. Here the owner of the k-th column block sends the column block packed together with the pivoting information to other processors.

Directed acyclic task graphs (DAGs) can be used to model irregular parallelism arising in this partitioned sparse LU program. Previous work on exploiting task parallelism for sparse Cholesky factorization has used elimination trees (e.g. [?, ?]), which is a good way to expose the available parallelism because the

for $k = 1$ **to** N
    Perform task $Factor(k)$;
    for $j = k + 1$ **to** N with $U_{kj} \neq 0$
        Perform task $Update(k, j)$;
    endfor
endfor

Figure 4.1: A partitioned sparse LU factorization with partial pivoting.

$Factor(k)$
    for $m = S(k)$ to $S(k + 1) - 1$
        Find the pivoting row t in column m;
        Interchange row t and row m of the column block k;
        Scale column m and update rest of columns in this column block;
    endfor

Figure 4.2: The description of task $Factor(k)$.

$Update(k, j)$
Interchange rows according to the pivoting sequence;
Let L_t be the lower triangular part of L_{kk};
if the submatrix U_{kj} is dense
$U_{kj} = L_t^{-1} * U_{kj}$;
else for each dense subcolumn c_u of U_{kj}
$c_u = L_t^{-1} * c_u$;
endfor

for each nonzero submatrix $A_{ij}, i > k$ in column block j
if the submatrix U_{kj} is dense
$A_{ij} = A_{ij} - L_{ik} * U_{kj}$;
else for each dense subcolumn c_u of U_{kj}
Let c_b be the corresponding dense subcolumn of A_{ij};
$c_b = c_b - L_{ik} * c_u$;
endfor
endfor

Figure 4.3: A description of task $Update(k, j)$.

input matrix is symmetric and pivoting is not involved. However by using DAGs, effective load balancing and graph scheduling algorithms can be applied to expose more parallelism. Given the task definitions in Figures 4.1, 4.2 and 4.3 we can define the structure of a sparse LU task graph in the following.

- There are N tasks $Factor(k)$, where $1 \leq k \leq N$.
- There is a task $Update(k, j)$ if $U_{kj} \neq 0$, where $1 \leq k < j \leq N$. For a dense matrix, there will be a total of $N(N-1)/2$ updating tasks.
- There is a dependence edge from $Factor(k)$ to task $Update(k, j)$, where $k < j$ and $U_{kj} \neq 0$.
- There is a dependence from $Update(k, k')$ to $Factor(k')$, where $k < k'$, $U_{kk'} \neq 0$ and there exists no task $Update(t, k')$ such that $k < t < k'$.
- There is a dependence from $Update(k, j)$ to $Update(k', j)$, where $k < k'$ and there exists no task $Update(t, j)$ such that $k < t < k'$ [2].

Figure 4.4(a) shows a nonzero pattern of a partitioned matrix . Figure 4.4(b) is the corresponding task dependence graph.

4.2 Data mapping

For block-oriented matrix computation, 1-D column-block mapping and 2-D block mapping are commonly used. The 1-D column-block mapping, also referred to as panel mapping in [?], views the processors as an $1 \times p$ grid, and a column-block is the minimum unit for data mapping.

[2] This is because of row interchange, but it is not always necessary. Here we take a conservative approach to simplify implementation. Our experience with Cholesky factorization with commuting $Update()$ tasks [?] indicates that the loss of performance is not substantial, about 6% in average when graph scheduling is used.

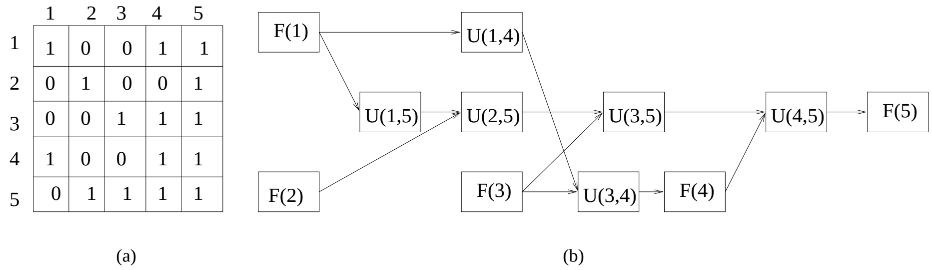

Figure 4.4: (a) A block pattern of partitioned matrix. (b) The dependence graph derived from the partitioning result. For convenience, $F()$ is used to denote $Factor()$, $U()$ is used to denote $Update()$.

A 2-D block mapping views the processors as a 2-D $p_r \times p_c$ grid, and a block is the minimum unit for data mapping. 2-D block mapping are usually considered more scalable than 1-D column-block mapping for general matrix computation, because it tends to have better computation load balance and lower complexity of communication volume. A common way for 2-D block mapping is cyclic mapping, in which a block A_{ij} is assigned to the processor $P_{i \ mod \ p_r, j \ mod \ p_c}$. This kind of mapping has following properties: a row block is assigned to a row of processors, and a column block is assigned to a column of processors as shown in Figure 4.5. In this chapter we discuss the algorithms by using 1-D data mapping schemes.

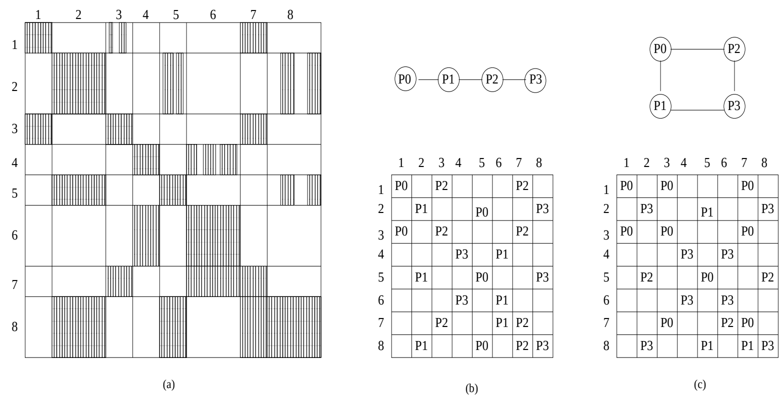

Figure 4.5: (a) A partitioned sparse matrix; b) A 1-D cyclic mapping of column blocks; (c) 2-D cyclic mapping of blocks.

In the 1-D data mapping, all submatrices, including both L and U part, of the same column block will reside in the same processor. Column blocks are mapped to processors in a cyclic manner or based on other scheduling techniques such as graph scheduling. Tasks are assigned based on owner-compute rule, i.e., tasks that modify the same column block are assigned to the same processor that owns the column block.

One disadvantage of 1-D data mapping is that it serializes the computation in a single $Factor(k)$ or $Update(k, j)$ task. In other words, a single $Factor(k)$ or $Update(k, j)$ task will be performed by one processor. But this mapping strategy has an advantage that both pivot searching and subrow interchange can be done locally without any communication. Another advantage is that parallelism modeled by the above dependence structure can be effectively exploited using graph scheduling techniques.

4.3 Scheduling and run-time support for 1-D methods

In this section we discuss how 1-D sparse LU tasks are scheduled and executed so that parallel time can be shortened. We use task dependence graphs to guide scheduling and have investigated two types of scheduling schemes.

- **Compute-ahead scheduling (CA).** This is to use block-cyclic mapping of tasks with a compute-ahead execution strategy, which is demonstrated in Figure 4.6. This idea has been used to speed up parallel dense factorizations [**?**]. It executes the numerical factorization layer by layer based on the current submatrix index. The parallelism is exploited for concurrent updating. In order to overlap computation with communication, the $Factor(k + 1)$ task is executed as soon as $Factor(k)$ and $Update(k, k + 1)$ (if exists) finish so that the pivoting sequence and column block $k + 1$ for the next layer can be communicated as early as possible.

- **Graph scheduling**. We order task execution within each processor using the graph scheduling algorithms in [**?**]. The basic optimizations are balancing processor loads and overlapping computation with communication to hide communication latency. These are done by utilizing global dependence structures and critical path information.

Graph scheduling has been shown effective in exploiting irregular parallelism for other applications (e.g. [**?**, **?**]). Graph scheduling should outperform the CA scheduling for sparse LU because it does not have a constraint in ordering $Factor()$ tasks. We demonstrate this point using the LU task graph in Figure 4.4. For this example, the Gantt charts of the CA schedule and the schedule derived by our graph scheduling algorithm are listed in Figure 4.7. It is assumed that each task has a computation weight 2 and each edge has communication weight 1. It is easy to see that our scheduling approach produces a better result than the CA schedule.

If we look at the CA schedule carefully, we can see that the reason is that CA can look ahead only one step so that the execution of task $Factor(3)$ is placed after $Update(1,5)$. On the other hand, the graph scheduling algorithm detects that $Factor(3)$ can be executed before $Update(1,5)$ which leads to better overlap of communication with computation.

However the implementation of the CA algorithm is much easier since the efficient execution of a sparse task graph schedule requires a sophisticated run-time system to support asynchronous communication protocols. We have used the RAPID run-time system [**?**] for the parallelization of sparse LU using graph scheduling. The key optimization is to use Remote Memory Access (RMA) to communicate a data object between two processors. It does not incur any copying/buffering during a data transfer since low communication overhead is critical for sparse code with mixed granularities. RMA is available in modern multi-

```
if column block 1 is local
    Perform task Factor(1);
    Broadcast column block 1 and the pivoting sequence;
for k = 1 to N − 1
    if column block k + 1 is local
        if U_{k,k+1} ≠ 0
            Receive column block k and the pivoting choices;
            Interchange rows according to the pivoting sequence;
            Perform task Update(k, k + 1);
        Perform task Factor(k + 1);
        Broadcast column block k+1 and the pivoting sequence;
    for j = k + 2 to N with U_{kj} ≠ 0
        if column block j is local
            if column block k has not been received
                Receive column block k and the pivoting choices;
            Interchange rows according to the pivoting sequence;
            Perform task Update(k, j);
    endfor
endfor
```

Figure 4.6: The 1-D code using compute-ahead schedule.

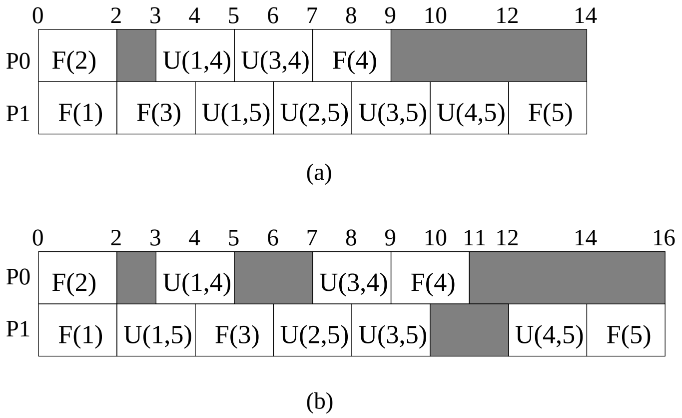

Figure 4.7: (a) A schedule derived by our graph scheduling algorithm. (b) A compute-ahead schedule. For convenience $F()$ is used to denote $Factor()$, $U()$ is used to denote $Update()$.

processor architectures such as Cray-T3D [?], T3E [?] and Meiko CS-2 [?]. Since the RMA directly writes data to a remote address, it is possible that the content at the remote address is still being used by other tasks and then the execution at the remote processor could be incorrect. Thus for a general computation, a permission to write the remote address needs to be obtained before issuing a remote write. However in the RAPID system, this hand-shaking process is avoided by a carefully designed task communication protocol [?]. This property greatly reduces task synchronization cost. As shown in [?], the RAPID sparse code can deliver more than 70% of the speedup predicted by the scheduler on Cray-T3D. In addition, using RAPID system greatly reduces the amount of implementation work to parallelize sparse LU.

4.4 Parallel performance of 1-D codes

In this section, we report a set of experiments conducted to examine the overall parallel performance of 1-D codes, the effectiveness of scheduling and supernode amalgamation.

Overall Performance: We list the MFLOPS numbers of the 1-D RAPID code obtained on various number of processors on Cray T3D and T3E for several testing matrices in Table 4.1 and 4.2. We know that the megaflops of `DGEMM` on T3E is about 3.7 times as large as that on T3D, and the RAPID code after using a upgraded machine is speeded up about 3 times in average, which is satisfactory. For the same machine, the performance of the RAPID code increases when the number of processors increases and speedups compared to the pure S^* sequential code (if applicable) can reach up to 17.7 on 64 T3D nodes and 24.1 on 64 T3E nodes. From 32 to 64 nodes, the performance gain is small except for matrices `goodwin`, `e40r0100` and `b33_5600`, which are much larger problems than the rest. The reason is that those small tested matrices do not have enough amount of computation and parallelism to saturate a large number of processors when the elimination process proceeds toward the end. It is our belief that better and more scalable performance can be obtained on larger matrices. But currently the available memory on each node of T3D or T3E limits the problem size that can be solved with the current version of the RAPID code.

Matrix	P=4	P=8	P=16	P=32	P=64
sherman5	25.8	40.8	53.8	64.9	68.4
lnsp3937	19.6	29.7	45.4	57.1	63.0
lns3937	17.6	28.5	43.5	50.3	52.1
sherman3	30.0	45.7	61.1	64.3	66.3
jpwh991	23.2	40.5	51.2	58.0	60.0
orsreg1	30.6	51.2	68.7	75.3	75.3
goodwin	54.0	87.9	136.4	182.0	218.1
b33_5600	80.4	135.7	217.6	341.8	405.3

Table 4.1: Absolute performance (MFLOPS) of the 1-D RAPID code on T3D.

Effectiveness of Graph Scheduling: We compare the performance of 1-D CA code with 1-D RAPID code in Figure 4.8. The Y axis is $1 - PT_{RAPID}/PT_{CA}$,

Matrix	P=4	P=8	P=16	P=32	P=64
sherman5	79.0	133.1	168.6	210.7	229.9
lnsp3937	62.1	93.8	145.5	183.5	201.0
lns3937	55.2	93.1	135.4	148.9	165.5
sherman3	90.7	143.5	192.8	199.0	212.7
jpwh991	75.6	124.2	173.9	193.2	217.3
orsreg1	90.6	160.3	215.6	223.3	231.6
goodwin	135.7	238.0	373.7	522.6	655.8
e40r0100	133.1	242.2	426.1	649.8	699.4
b33_5600	199.0	365.0	642.0	1029.0	1353.2

Table 4.2: Absolute performance (MFLOPS) of the 1-D RAPID code on T3E.

where PT stands for parallel time. For 2 and 4 processors, in certain cases, the compute-ahead code is slightly faster than the RAPID code. But for the number of processors more than 4, the RAPID code runs $10 - 72\%$ faster. The more processors involved, the bigger the performance gap tends to be. The reason is that for a small number of processors, there are sufficient tasks making all processors busy and the compute-ahead schedule performs well while the RAPID code suffers a certain degree of system overhead. For a larger number of processors, schedule optimization becomes important since there is limited parallelism to exploit.

Effectiveness of supernode amalgamation: We have examined how effective our supernode amalgamation strategy is using the 1-D RAPID code. Let PT_a and PT be the parallel time with and without supernode amalgamation respectively. The parallel time improvement ratios $(1 - PT_a/PT)$ on T3E for several testing matrices are listed in Table 4.3 and similar results on T3D are in [**?**]. Apparently the supernode amalgamation has brought significant improvement due to the increase of supernode size which implies an increase of the task granularities. This is important to obtaining good parallel performance [**?**].

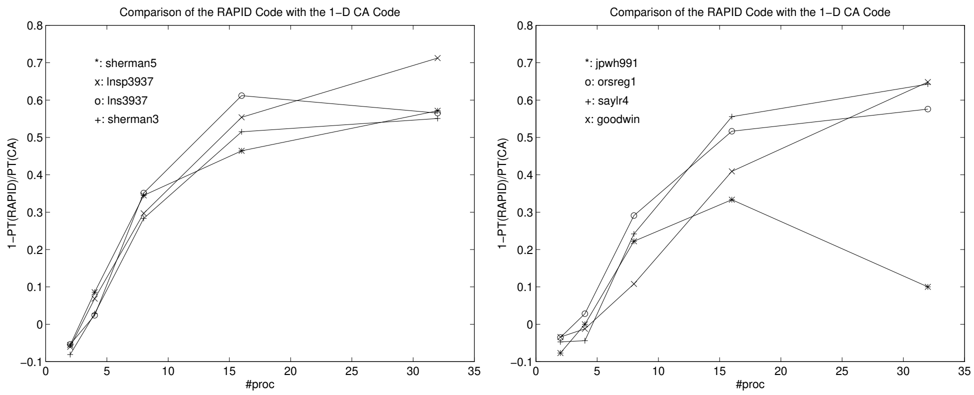

Figure 4.8: Impact of different scheduling strategies on 1-D code approach.

Matrix	P=1	P=2	P=4	P=8	P=16	P=32
sherman5	47%	47%	46%	50%	40%	43%
lnsp3937	50%	51%	53%	53%	51%	39%
lns3937	55%	54%	54%	54%	51%	35%
sherman3	20%	25%	23%	28%	22%	14%
jpwh991	48%	48%	48%	50%	47%	40%
orsreg1	16%	18%	18%	26%	15%	10%
saylr4	21%	22%	23%	23%	30%	18%

Table 4.3: Parallel time improvement obtained by supernode amalgamation.

4.5 Conclusions

In this chapter we presented the parallel sparse *LU* factorization with 1-D data mapping schemes. Two different scheduling strategies have been used. One is compute-ahead scheduling, which has simple run-time control but suffers from poor performance. The other one uses sophisticated graph scheduling and uses RAPID system for run-time support, which achieves much higher performance than using compute-ahead scheduling.

Though the performance results of the 1-D RAPID code are promising, it does not scale very well on large number of processors, because it does not expose sufficient parallelism. In 1-D data mapping, a $Factor()$ task is performed on a single processor, which bounds the speedup of the 1-D approaches. In addition, the communication volume of 1-D algorithms has complexity of $O(p)$, so the length of critical path increases quickly along with the number of processors. These make the 1-D approaches do not scale very well in terms of time.

Another issue is that a time-efficient scheduling may not be space-efficient. Specifically, to support concurrency among multiple updating stages in both RAPID and CA codes, multiple buffers are needed to keep pivoting column blocks of different stages on each processor. Therefore for a given problem, the per processor space complexity of the 1-D codes could be as high as $O(S_1)$, where S_1 is the space complexity for a sequential algorithm. For sparse LU, each processor in the worst case may need a space for holding the entire matrix for out-of-order execution. The RAPID system [?] also needs extra memory space to hold dependence structures. These limit the problems which can be solved by 1-D codes. In the next chapter, we will present an algorithm with 2-D data mapping, which is more scalable in terms of time as well as space and have potential to solve very large sparse matrices.

Chapter 5

Parallel Sparse LU Factorization with 2-D Data Mapping

In this chapter, we discuss the parallel sparse LU factorization with partial pivoting and 2-D data mapping. The distinguished features of this algorithm are employing asynchronous scheduling to expose irregular parallelism while maintaining low memory requirements. These features make the algorithm be time and space efficient. It has achieved dramatic performance for sparse LU factorization on distributed memory machines. The rest of the chapter is organized as follows. Section 5.1 presents the 2-D data mapping scheme used in the algorithm. Section 5.2 presents the task partitioning and scheduling scheme of the 2-D approach. Section 5.3 analyzes available parallelism for the 2-D approach. Section 5.4 analyzes the memory requirements of the algorithm. Section 5.5 presents the performance of the 2-D approach on Cray T3D and T3E. Section 5.6 compares the performance of 2-D approach and 1-D approaches with a set of medium size benchmark matrices. Section 5.7 demonstrates the efficiency of the asynchronous scheduling. Section 5.8 points out some possible improvements for the 2-D code. Section 5.9 concludes the section and summarizes the contributions.

5.1 2-D data mapping

2-D data mapping has been shown more scalable than 1-D mapping for sparse Cholesky in [?, ?], because in 2-D mapping a block is the minimum unit for data mapping, which results in finer grained tasks and better load balancing on large number of processors, and 2-D mapping has lower order of communication volume. However there are several difficulties to apply the 2-D block-oriented mapping to the case of sparse LU factorization even the static structure is predicted. Firstly, pivoting operations and row interchanges require frequent and well-synchronized interprocessor communication when submatrices in the same column block are assigned to different processors. Effective exploitation of irregular parallelism in the 2-D case requires a highly efficient asynchronous execution mechanism and a delicate message buffer management. Secondly, the task dependences in sparse LU with 2-D mapping is too complicated to utilize and schedule all possible irregular parallelism from sparse LU. Lastly, managing low space complexity is hard since exploiting irregular parallelism to a substantial degree may need more buffer space.

Our 2-D algorithm uses a standard mapping scheme. In this scheme, p available processors are viewed as a two dimensional grid: $p = p_r \times p_c$. A nonzero submatrix block A_{ij} (could be an L block or a U block) is assigned to processor $P_{i \bmod p_r,\, j \bmod p_c}$. This scheme has the feature that a column block is assigned to a column of processors, and a row block is assigned to a row of processors, so a single $Factor(k)$ or $Update(k,j)$ task defined in Chapter 4 can be parallelized on p_r processors, and communication only happens within a processor row or a processor column. That is the major reason why 2-D mapping shall be more scalable than 1-D mapping in terms of time efficiency. In the algorithm all data objects including matrix structure are distributed onto multiple processors, and the message buffers are very miserly regardless of number of processors while exposing parallelism substantially. These make the algorithm be scalable in terms of space efficiency.

5.2 An asynchronous scheduling algorithm

In this section, we present an asynchronous 2-D algorithm which can substantially overlap multi-stages of updating but its memory requirements are much smaller than those of 1-D methods. Figures 5.1 shows the main control of the algorithm in an SPMD style. Figure 5.2 shows the SPMD code for $Factor(k)$ which is executed by processors of column $k \bmod p_c$. Also we divide the function of $Update()$ (in Figure 4.3) into two parts: $ScaleSwap()$ which does scaling and delayed row interchange for submatrix $A_{k:N,\ k+1:N}$ as shown in Figure 5.3; $Update\_2D()$ which does submatrix updating as shown in Figure 5.4. Note that in these task descriptions, unlike the conventional task definition, each task is performed on a column of processors, and there is communication within a task. In all the figures, the statements which involve interprocessor communication are marked with $*$.

It can be seen that the **computation flow** of this 2-D code is still controlled by the pivoting tasks $Factor(k)$. The order of execution for $Factor(k)$, $k = 1, 2, \cdots, N$ is sequential. Because each $Factor(k)$ task is parallelized on p_r processors, and because compute-ahead scheduling is used, i.e., $Factor(k+1)$ is executed as soon as $Update\_2D(k,\ k+1)$ finishes, the serial execution of $Factor(k)$ tasks shall not dominate the critical path. $Update\_2D()$ tasks, where most of the computation comes from, can execute in parallel among all processors. The asynchronous parallelism of $Update\_2D()$ tasks comes from two levels. First a single stage of tasks $Update\_2D(k,\ k+1:N)$ can be executed concurrently on all processors. In addition, different stages of $Update\_2D()$ tasks from $Update\_2D(k,\ k+1:N)$ and $Update\_2D(k',\ k'+1:N)$, where $k \neq k'$, can also be overlapped.

Some detailed explanation for pivoting, scaling and swapping is given below.

```
Let (my_rno, my_cno) be the 2-D coordinates of this processor;
if my_cno = 1 then Perform task Factor(1);
for k = 1 to N − 1
    Perform ScaleSwap(k);
    if my_cno = (k + 1) mod p_c
        Perform Update_2D(k, k + 1);
        Perform Factor(k + 1);
    for j = k + 2 to N
        if my_cno = j mod p_c
            Perform Updatej_2D(k, j);
    endfor
endfor
```

Figure 5.1: The SPMD code of 2-D asynchronous code.

In line (5) of Figure 5.2, the whole subrow is communicated when each processor reports its local maximum to $P_{k \bmod p_r,\, k \bmod p_c}$, i.e., the processor that owns the L_{kk} block. Let m be the current global column number on which the pivoting is conducted, then without further synchronization, processor $P_{k \bmod p_r,\, k \bmod p_c}$ can locally swap subrow m with subrow t which contains the selected pivoting element. This shortens the waiting time to conduct further updating with a little more communication volume. However in line (08), processor $P_{k \bmod p_r,\, k \bmod p_c}$ must send the original subrow m to the owner of subrow t for swapping, and the selected subrow t to other processors as well for updating. In $Factor()$ tasks, synchronizations take place at lines (05), (07) and (08) when each processor reports its local maximum to $P_{k \bmod p_r,\, k \bmod p_c}$, and $P_{k \bmod p_r,\, k \bmod p_c}$ broadcasts the subrow containing global maximum along the processor column. At line (12) each processor in column $k \bmod p_c$ broadcasts the pivot sequence along its own processor row rather than processor $P_{k \bmod p_r,\, k \bmod p_c}$ broadcasts to all processors. That is important because it ensures that the communication complexity in the critical path is $O(max(p_r, p_c))$ instead of $O(p)$. At line (14) of $Factor()$, and at line (05) and (10) of $ScaleSwap()$, the small messages are packed together to minimize communication startup overhead.

5.3 Parallelism exploitation

We examine the degree of parallelism exploited in this algorithm by determining number of updating stages that can be overlapped. Using this information we can also determine the extra buffer space needed per processor to execute this algorithm effectively. We define the **stage overlapping degree** for updating tasks as

$$max\{|k - k'| \mid Update\_2D(k,*) \text{ and } Update\_2D(k',*) \text{ concur.}\}$$

Here $Update\_2D(k,*)$ denotes a set of $Update\_2D(k,j)$ tasks where $k < j \leq N$.

(01) $Factor(k)$
(02) **for** $m = S(k)$ to $S(k+1)-1$
(03) Find out local maximum element of column m;
(04) **if** $my\_rno \neq k \bmod p_r$
(05)* Send the subrow within column block k containing the local maximum to processor $P_{k \bmod p_r,\ k \bmod p_c}$;
(06) **if** this processor owns L_{kk}
(07)* Collect all local maxima and find the pivot row t;
(08)* Broadcast the subrow t within column block k along this processor column and interchange subrow t and subrow m if necessary.
(09) Scale local entries of column m;
(10) Update local subcolumns from column $m+1$ to $S(k+1)-1$;
(11) **endfor**
(12)* Multicast the pivot sequence along this processor row;
(13)* **if** this processor owns L_{kk} **then** Multicast L_{kk} along this processor row;
(14)* Multicast the part of nonzero blocks in $L_{k+1:N,\ k}$ owned by this processor along this processor row;

Figure 5.2: Parallel execution of $Factor(k)$ for the 2-D asynchronous code.

$ScaleSwap(k)$

if $my\_cno \neq k \bmod p_c$ **then** receive the pivot sequence from $P_{my\_rno,\ k \bmod p_c}$;

for $m = S(k)$ to $S(k+1) - 1$

if This processor own a part of row m or the pivot row t for column m

* Interchange nonzero parts of row t and row m owned by this processor;

endfor

if $my\_rno = k \bmod p_r$

* **if** $my\_cno \neq k \bmod p_c$ **then** receive L_{kk} from $P_{my\_rno,\ k \bmod p_c}$;

Scale nonzero blocks in $U_{k,\ k:N}$ owned by this processor;

* Multicast the scaling results along this processor column;

* **if** $my\_cno \neq k \bmod p_c$ **then** receive $L_{k:N,\ k}$ from $P_{k \bmod p_r,\ my\_cno}$;

* **if** $my\_rno \neq k \bmod p_r$ **then** receive $U_{k,\ k:N}$ from $P_{my\_rno,\ k \bmod p_c}$;

Figure 5.3: Task $ScaleSwap(k)$ for the 2-D asynchronous code.

```
Update_2D(k, j)
    if j mod p_c = my_cno
      for i = k + 1 to N
        if i mod p_r = my_rno
          Update A_ij using L_ik and U_kj;
      endfor
```

Figure 5.4: *Update_2D*(k, j) for the 2-D asynchronous code.

Theorem 4 *For the asynchronous 2-D algorithm on p processors where $p > 1$ and $p = p_r \times p_c$, the reachable upper bound of overlapping degree is p_c among all processors; and the reachable upper bound of overlapping degree within a processor column is $min(p_r - 1, p_c)$.*

Proof: We will use the following facts in proving the theorem:

- **Fact 1.** $Factor(k)$ is executed at processors with column number k mod p_c. Processors on this column are synchronized. When a processor completes $Factor(k)$, this processor can still do *Update_2D*$(k-1, *)$ as shown in Figure 5.2, but all *Update_2D*$(k-t, *)$ tasks belonging to this processor where $t > 1$ must have been completed on this processor.

- **Fact 2.** $ScaleSwap(k)$ is executed at processors with row number k mod p_c. When a processor completes $ScaleSwap(k)$, all *Update_2D*$(k-t, *)$ tasks belonging to this processor where $t > 0$ must have been completed on this processor.

Part 1. First we show that *Update_2D*$()$ tasks can be overlapped to a degree of p_c among all processors.

When $p_c = 1$, it is trivial based on Fact 1. When $p_c > 1$, we can imagine a scenario in which all processors in column 0 have just finished task $Factor(k)$, and some of them are still working on $Update\_2D(k-1,*)$ tasks, and processors in column 1 could go ahead and execute $Update\_2D(k,*)$ tasks. After processors in column 1 finish $Update\_2D(k,k+1)$ task, they will execute $Factor(k+1)$. Then after finishing $Update\_2D(k,*)$ tasks, processors in column 2 could execute $Update\_2D(k+1,k+2)$ task and start $Factor(k+2)$ task and so on. Finally, processors in column p_c-1 could execute $Factor(k+p_c-1)$ and then start $Update\_2D(k+p_c-1,*)$. At this moment, processors in column 0 may be still working on $Update\_2D(k-1,*)$. Thus the overlapping degree is p_c.

Now we will show by contradiction that the maximum overlapping degree is p_c. Assume that at some moment, there exist two updating stages being executed concurrently: $Update\_2D(k,*)$ and $Update\_2D(k',*)$, where $k' > k + p_c$. Then $Factor(k')$ must have been completed. Without loss of generality, assuming that processors in column 0 execute $Factor(k')$, then according to Fact 1 all $Update\_2D(j,*)$ tasks where $j < k'-1$ should be completed before this moment. Since block cyclic mapping is used, it is easy to see each processor column has performed one of the $Factor(j)$ tasks where $k'-p_c+2 \leq j \leq k'$. Then $Update\_2D(k'-p_c+1)$ should be completed on all processors. Then for any concurrent stage $Update\_2D(k,*)$, k must satisfy $k \geq k'-p_c$, which is a contradiction.

Part 2. First, we show that overlapping degree $min(p_r-1,p_c)$ can be achieved within a processor column. For the convenience of illustration, we consider a scenario in which all delayed row interchanges in $ScaleSwap()$ take place locally without any communication within a processor column. Therefore there is no interprocessor synchronization going on within a processor column except in $Factor()$ tasks. Assuming $p_r-1 < p_c$, we can imagine at some moment, processors in column 0 have completed $Factor(s)$, and $P_{0,0}$ has just finished $ScaleSwap(s)$, and starts executing $Update\_2D(s,*)$, where $s \bmod p_r = 0$ and $s \bmod p_c = 0$. Then processors in column 1 will execute $Update\_2D(s,s+1)$ and then $Factor(s+1)$, after which $P_{1,0}$ can start $ScaleSwap(s+1)$ and then

$Update\_2D(s+1,*)$. Following this reasoning, after $Update\_2D(s+p_r-2, s+p_r-1)$ and $Factor(s+p_r-1)$ have been finished on processors of column p_r-1, $P_{p_r-1,0}$ could complete previous $Update\_2D()$ tasks and $ScaleSwap(s+p_r-1)$, and start $Update\_2D(s+p_r-1,*)$. Now $P_{0,0}$ may be still working on $Update\_2D(s,*)$. Thus the overlapping degree is p_r-1. If $p_r \geq p_c+1$, obviously the above reasoning will stop when processors of column p_c-1 finish $Update\_2D(s+p_c-2, s+p_c-1)$ and $Factor(s+p_c-1)$. In that case when $P_{p_c-1,0}$ is to start $Update\_2D(s+p_c-1,*)$, the processor $P_{p_r-1,0}$ could be still working on $Update\_2D(s-1,*)$ because of the compute ahead scheduling. Hence the overlapping degree is p_c.

Now we need to show that the upper bound of overlapping degree within a processor column is $min(p_r-1, p_c)$. We have already shown in the proof of Part 1 that the overall overlapping degree is less than p_c, so is the overlapping degree within a processor column. To prove it is also less than p_r-1, we can use the similar proof as that for part 1, except using $ScaleSwap(k)$ to replace $Factor(k)$, and using Fact 2 instead of Fact 1.

■

5.4 Memory requirement analysis

Knowing degree of overlapping is important in determining the amount of memory space needed to accommodate those communication buffers on each processor for supporting asynchronous execution. Buffer space is additional to data space needed to distribute the original matrix. There are four types of communication that needs buffering:

1. Pivoting along a processor column (lines (05), (07), and (08) in Figure 5.2), which includes communicating pivot positions and multicasting pivot rows. We call the buffer for this purpose *Pbuffer*.

2. Multicasting along a processor row (line (12), (13) and (14) in Figure 5.2). 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 (line (05) in Figure 5.3). We call this buffer *Ibuffer*.

4. Multicasting along a processor column (line (10) in Figure 5.3). The data includes local nonzero blocks of a row panel. We call the buffer *Rbuffer*.

Here we assume that $p_r \le p_c + 1$, because based on our experimental results, setting $p_r \le p_c + 1$ always leads to better performance. Thus the overlapping degree of $Update\_2D()$ tasks within a processor row is at most p_c, and the overlapping degree within a processor column is at most $p_r - 1$. Then we need p_c separate *Cbuffer*'s for overlapping among different columns and $p_r - 1$ separate *Rbuffer*'s for overlapping among different rows.

We estimate the size of each *Cbuffer* and *Rbuffer* as follows. Assuming that the sparsity ratio of a given matrix is s after fill-in and the maximum block size is $BSIZE$, each *Cbuffer* is of size:

$$C = max\{\text{space for local nonzero blocks of } L_{k:N,k}, 1 \le k \le N\} \approx n{\cdot}BSIZE{\cdot}s/p_r.$$

Similarly each *Rbuffer* is of size:

$$R = max\{\text{space for local nonzero blocks of } U_{k,k+1:N}, 1 \le k \le N\} \approx n{\cdot}BSIZE{\cdot}s/p_c.$$

We ignore the buffer size for *Pbuffer* and *Ibuffer* because they are very small (the size of *Pbuffer* is only about $BSIZE \cdot BSIZE$ and the size of *Ibuffer* is about $s \cdot n/p_c$). Thus the total buffer space needed for the asynchronous execution is: $C \cdot p_c + R \cdot (p_r - 1) \approx n \cdot BSIZE \cdot s \cdot (p_c/p_r + p_r/p_c)$.

Notice that the sequential space complexity $S_1 = n^2 s$. In practice, we set $p_c/p_r = 2$. Therefore the buffer space complexity for each processor is $2.5 \cdot n \cdot$

$BSIZE \cdot s$, or $\frac{2.5 \cdot BSIZE}{n} \cdot S_1$, which is very small for a large matrix. For all the benchmark matrices we have tested, the buffer space is less than 100 K words. Given a sparse matrix, if the matrix data is evenly distributed onto p processors, the total memory requirement per processor is $S_1/p+O(1)$ considering $n \gg p$ and $n \gg BSIZE$. This leads us to conclude that the 2-D asynchronous algorithm is very space scalable.

5.5 2-D code performance

As mentioned before, our 2-D code exploits more parallelism but maintains a lower space complexity, and has much more potential to solve large problems. We show the absolute performance obtained for some large matrices on T3D in Table 5.1. Since some matrices cannot fit for a small number of processors, we only list results on 16 or more processors. The maximum absolute performance achieved on 64 nodes of T3D is 1.48 GFLOPS, which is translated to 23.1 MFLOPS per node. For 16 nodes, the per-node performance is 32.8 MFLOPS.

	P=16		P=32		P= 64	
Matrix	Time	Mflops	Time	Mflops	Time	Mflops
goodwin	6.0	110.7	4.6	145.2	3.6	184.8
e40r0100	21.2	116.6	15.2	162.5	10.5	235.4
fidap011	87.9	305.0	53.4	501.8	33.4	802.6
raefsky4	129.8	242.9	76.0	413.8	43.2	719.2
vavasis3	169.8	525.4	100.8	885.1	60.3	1480.2

Table 5.1: Performance results of the 2-D code for large matrices on T3D.

Table 5.2 shows the performance numbers on T3E for the 2-D code. We have achieved up to 6.88 GFLOPS on 128 nodes. For 64 nodes, megaflops on T3E are from 3.1 to 3.4 times as large as that on T3D. Again considering that

DGEMM megaflops on T3E is about 3.7 times as large as that on T3D, our code performance after using a upgraded machine is good.

Matrix	P=8		P=16	P=32	P=64	P=128
	Time	Mflops	Mflops	Mflops	Mflops	Mflops
goodwin	3.1	215.2	344.6	496.3	599.2	715.2
e40r0100	12.1	205.1	342.9	515.8	748.0	930.8
fidap011	50.7	489.3	878.1	1524.3	2504.4	3828.5
raefsky4	79.4	391.2	718.9	1290.7	2233.3	3592.9
inaccura	16.8	244.6	415.2	655.8	1048.0	1391.4
af23560	22.3	285.4	492.9	784.3	1123.2	1512.7
vavasis3	109.7	813.4	1519.0	2651.9	4505.5	6878.1

Table 5.2: Performance results of the 2-D code on T3E.

5.6 Performance comparison of 1-D and 2-D codes

Notice that 1-D codes cannot solve the larger matrices of Table 5.2. For some small and medium size matrices solvable by both 1-D RAPID and 2-D codes, we compare the parallel time differences by computing the ratio: $1 - PT_{RAPID}/PT_{2D}$ and the result is in Figure 5.5. For these matrices the 1-D RAPID code achieves better performance because it uses sophisticated graph scheduling technique to guide the mapping of column blocks and ordering of tasks, which results in better overlapping of communication with computation. The performance difference is larger for the matrices listed in the left of Figure 5.5 compared to the right of Figure 5.5. We partially explain the reason by analyzing load balance factors of the 1-D RAPID code and the 2-D code in Figure 5.6. The load balance factor is defined as $work_{total}/(P \cdot work_{max})$ [?]. Here we only count the work from the updating part because it is the major part of the computation. The 2-D code has better load balance, which can make up for the impact of lacking of efficient task scheduling. This is verified by Figure 5.5 and Figure 5.6. One can see that when the load balance factor of the 2-D code is close to that of the RAPID code (e.g., `lnsp3937`), the performance of the RAPID code is much better than the 2-D code; when the load balance factor of the 2-D code is significantly better than that of the RAPID code (e.g., `jpwh991` and `orserg1`), the performance differences are smaller.

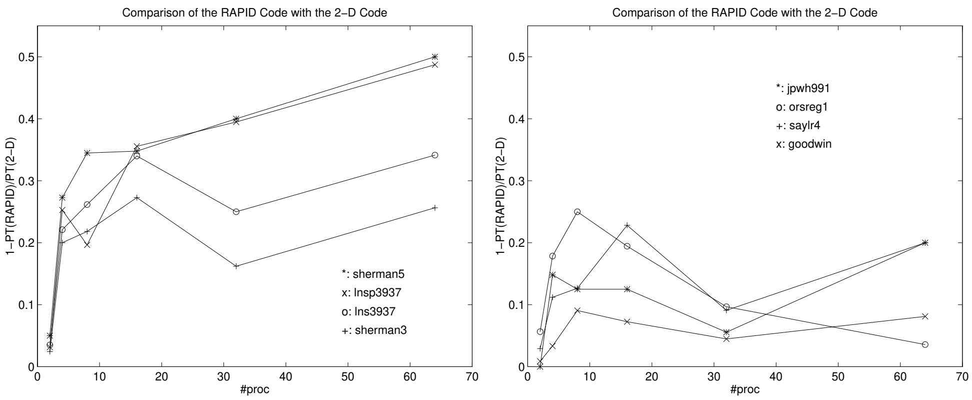

Figure 5.5: Performance comparison of 1-D and 2-D codes: $1 - PT_{1D}/PT_{2D}$.

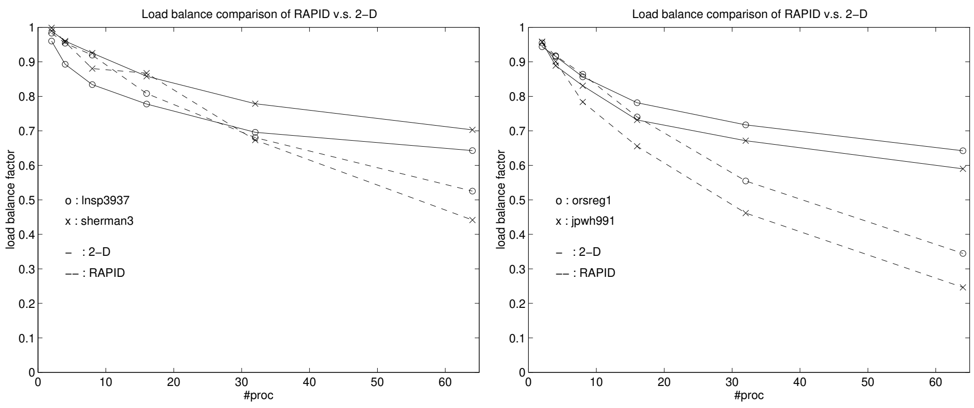

Figure 5.6: Comparison of load balance factors of 1-D RAPID code and 2-D code.

5.7 Impact of asynchronous scheduling

Using a global barrier in the 2-D code at each elimination step can simplify the implementation, but it cannot overlap computations among different updating stages. We have compared parallel time reductions between the asynchronous code and the synchronous code for some testing matrices in Table 5.3. It shows that asynchronous design improves performance significantly, especially on large number of processors on T3E. It demonstrates the importance of exploiting parallelism using asynchronous execution. The experiment data on T3D is in [**?**].

Matrix	P=2	P=4	P=8	P=16	P=32	P=64
sherman5	7.7%	6.4%	19.4%	28.1%	25.9%	24.1%
lnsp3937	7.3%	7.1%	22.2%	28.57%	26.9%	27.9%
lns3937	6.6%	2.8%	18.8%	26.5%	28.6%	26.8%
sherman3	10.2%	12.4%	20.3%	22.7%	26.0%	25.0%
jpwh991	8.7%	10.0%	23.8%	33.3%	35.7%	28.6%
orsreg1	6.1%	7.7%	17.5%	28.0%	20.5%	28.2%
saylr4	8.0%	10.7%	21.0%	29.6%	30.2%	27.4%
goodwin	5.4%	14.1%	14.2%	24.6%	26.0%	30.2%
e40r0100	2.4%	8.7%	8.1%	16.8%	18.1%	29.9%
fidap011	-	9.0%	6.9%	14.9%	12.6%	24.5%
raefsky4	-	9.4%	8.1%	16.2%	13.5%	27.1%
vavasis3	-	-	7.8%	17.4%	15.2%	29.0%

Table 5.3: Performance improvement of 2-D asynchronous code over 2-D synchronous code.

5.8 Possible improvements

We have considered five ways to further improve the performance of the 2-D approach. In this section, we outline the ideas for guiding the future research in sparse *LU*.

First of all, a $ScaleSwap(k)$ task can be divided into two parts and reordered to reduce the critical path. In current implementation, a $ScaleSwap(k)$ task applies row interchange to column blocks $k+1$ to N and scales the whole row of $\boldsymbol{U_k}$. This enables us to aggregate the small messages in $ScaleSwap(k)$ into larger ones. However, it also causes a serial execution order of $Factor(k)$, $ScaleSwap(k)$, $Update(k, k+1)$, $Factor(k+1)$, $\cdots$ Although task $ScaleSwap(k)$ contributes a small amount of computation, we have noticed that it induces high communication volume and frequent data movement due to poor locality in swapping, which may limit the scalability of the algorithm. The problem can be circumvented by dividing $ScaleSwap(k)$ into $ScaleSwap(k, k+1)$, and $ScaleSwap(k, k+2)$, where $ScaleSwap(k, k+1)$ applies row interchange only to column $k+1$ and scales $\boldsymbol{U_{k,k+1}}$, and $ScaleSwap(k, k+2)$ modifies the column blocks $k+2$ to N, and computing $Update(k, k+1)$ as soon as $ScaleSwap(k, k+1)$ is done. After this modification, the serial execution order will be shortened to $Factor(k)$, $ScaleSwap(k, k+1)$, $Update(k, k+1)$, $Factor(k+1)$ $\cdots$ and hence the algorithm shall be more scalable on large number of processors.

Secondly, after the first optimization, the overlapping of multiple $Factor()$ tasks is possible by utilizing supernode elimination trees. A simple approach is to overlap the computation of $Factor(k)$ and $Factor(k+1)$ if $parent(k) \neq k+1$. Potentially it will exploit more parallelism for sparser matrices.

Thirdly, heuristic algorithms can be used to improve computational load balancing. Our experimental studies show that 2-D block-cyclic mapping delivers good load balancing for a broad range of sparse matrices, but it does not consistently deliver good balancing for *all* matrices. In [**?**] a heuristic algorithm is used to improve load balancing for Cholesky factorization with 2-D data mapping. In the algorithm the mapping scheme has the property that a column block resides in a processor column, and a row block resides in a processor row. This property makes the algorithm still have a low communication volume, but less overlapping of multiple stages of $Update\_2D()$ tasks after applying the load balancing algorithm. A heuristic algorithm has to be developed by considering both load balancing and overlapping of computation at the same time.

Fourthly, threshold pivoting strategy could reduce interprocessor communication and provides better locality. In 1-D mapping, the row interchange are performed locally, therefore partial pivoting is the best choice because it provides good numerical stability. However in 2-D mapping, because the pivoting operations involve interprocessor communications, using threshold pivoting could reduce communication volume and communication times. Our preliminary study has shown that using threshold pivoting could improve the performance by $10-20\%$ for the 2-D code without compute-ahead scheduling. In [**?**] it has been shown that threshold pivoting delivers competitive numerical stability as partial pivoting. When numerical stability is not the most critical issue, for example in Newton's method, using threshold pivoting could shorten the total execution time, especially on systems with low communication bandwidth.

Lastly, the buffer space can be made even smaller by limiting stage overlapping degree of $Update\_2D()$ tasks if memory is scarce. In Section 5.4, we have mentioned that the buffer space is determined by matrix size and stage overlapping degree, and the space is a small constant compared to S_1 regardless of number of processors. However for very large matrices and very large p, it is also possible that the buffer space will be larger than S_1/p which limits the space scalability of the algorithm. This problem can be solved by controlling available overlapping degree. In the implementation we can control that the maximum stage overlapping degree among all processors to be p_c/t, and the degree within a processor column to be $min(p_r-1, p_c)/t$, where $1 \leq t \leq min(p_r-1, p_c)$. In the case, the complexity of the buffer space will drop to $\frac{BSIZE\cdot(p_c/p_r+p_r/p_c)}{n\cdot t}S_1$. With this optimization, the algorithm will be more space scalable.

5.9 Conclusions and contributions

In this chapter, we presented an asynchronous approach for sparse LU factorization with partial pivoting and 2-D data mapping. This algorithm employs asynchronous scheduling technique, which could overlap multiple stages of $Update\_2D()$ tasks significantly. By carefully designing communication buffers, the algorithm has low memory requirements per node, which enables the algorithm to solve very large problems. These features make the algorithm be both time and space efficient. To our best knowledge, it is the first algorithm for sparse LU factorization with 2-D block mapping, and it is also the first time and space efficient algorithm for sparse LU on distributed memory machines.

The approach has achieved breakthrough performance for sparse LU. On 128 nodes of Cray T3E, it achieves 6.88 GFLOPS for the matrix vavasis3, and it consistently delivers high performance for many large benchmark matrices. Considering that the previously highest performance on distributed memory machines for sparse LU is 1.35 GFLOPS by 1-D RAPID code on 64 processors of Cray T3E, this performance result makes a significant contribution to the sparse LU research. We have also observed that the highest performance achieved on shared memory machines by SuperLU is 2.58 GFLOPS on 8 nodes of Cray C90; and for the same CPU type and the same matrix vavasis3, SuperLU achieves 724 MFLOPS on 8 nodes of DEC AlphaServer 8400, and the 2-D approach achieves 813 MFLOPS on 8 nodes of Cray T3E. This leads us to conclude that the research of sparse LU factorization on distributed memory machines is very promising.

Chapter 6

Parallel Triangular Solves and Newton's Method

In this chapter, we discuss some issues about parallel triangular solves and parallel Newton's method for solving nonlinear systems. In Section 6.1 we introduce some background information about these problems. In Section 6.2 and 6.3 we discuss how to implement triangular solve by using the same data mapping schemes for the factorize phase so that the data shuffling can be avoided. In Section 6.4 we present how to compute forward substitution during factorization to reduce communication overhead. Section 6.5 demonstrates how our factorization and triangular solve kernel can be used in parallel Newton's method for solving nonlinear systems. Section 6.6 concludes the chapter, and point out the future research directions for these topics.

6.1 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 attentions [**?**] 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 [**?**], the triangular solves are executed multiple times by reusing the factor results. In these kinds of systems triangular solve might involve comparable amount of computation as 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 an algorithm for forward substitution, and give a theoretical analysis for the algorithm. The algorithm for back substitution is also presented.

Newton's method is important in solving nonlinear systems in scientific and engineering computations. However current existing software packages are disappointing to the users because of inefficient implementations of kernels. We have conducted some research on parallel Newton's method by using our sparse LU factorization kernel and the on-line triangular solves. We discuss some implementation issues of the parallel Newton's method in section 6.5.

6.2 Parallel triangular solve with 1-D data mapping

In this section we discuss the algorithm for triangular solve corresponding to parallel LU factorization with 1-D data mapping. Recall that in the LU factorization with 1-D data mapping, a column block (including both L and U part) resides in the same processor. Thus the data mapping of the triangular solve has to follow the same scheme to avoid excessive communication. We discuss two different program partitions for triangular solves, and compare their dependence patterns in the following subsections.

6.2.1 Forward substitution

Column oriented forward substitution

In Chapter 2 we presented a sequential algorithm for block sparse forward substitution. In 1-D data mapping, because L is column block oriented, the implementation is straightforward when defining a task $ForSub(k)$ as one step of the outer loop. The parallel algorithm following this task partitioning is shown in Figure 6.1, in which a task $ForSub(k)$ is defined as lines (02) to (08). In this algorithm we assume that b is replicated on all processors.

However, in this algorithm there is a true dependence from task $ForSub(k)$ to task $ForSub(k+1)$. At line (06) of task $ForSub(k)$ the computations are

```
for k = 1 to N
        b = P_k b;
        if column block k is owned by this processor
              Solve L_kk B_k = B_k;
              for i = k + 1 to n with L_ik ≠ 0
                    B_i = B_i - L_ik · B_k;
              endfor
              Send b[S(k) : n] to the owner of column block k + 1;
endfor
```

Figure 6.1: Column oriented block forward substitution. y is stored in b at output. B_i denotes $b_{S(i):S(i+1)-1}$.

commutable but column block k resides in a single processor in 1-D data mapping, so the execution is serialized. One may divide b into N objects and send B_i as soon as it has been modified in task $ForSub(k)$. This relaxes the dependence from task $ForSub(k)$ to $ForSub(k+1)$ and overlap the two tasks a little bit more. However it is difficult to overlap multiple $ForSub()$ tasks. To parallelize forward substitution for 1-D data mapping, we have to reorder the algorithm so that computation can be overlapped.

Row oriented forward substitution

In designing parallel algorithms, *loop interchange* has been widely used. However this strategy can not be directly applied to sparse forward substitution because there is a true dependence from step k to step $k+1$. In order to apply loop interchange, we have to remove the task dependences from step k to $k+1$.

The key idea is to move the permutation of b out of the main loop, and "permute" L dynamically. This results in the triangular system

$$\boldsymbol{L_D y} = \boldsymbol{P} b,$$

where $\boldsymbol{L_D}$ is the lower triangular factor of the conventional Gaussian elimination, and

$$\boldsymbol{P} = \boldsymbol{P_N P_{N-1} \cdots P_1}.$$

This is the block forward substitution for conventional Gaussian elimination with partial pivoting. The task dependences in this version are much less than those in the previous version. After this reordering, loop interchange can be applied to the algorithm to exploit more parallelism. However the main issue is how we can dynamically "permute" L without introducing overhead of memory management.

It can be seen that

$$\boldsymbol{P_N P_{N-1} \cdots P_{k+1} L_k} = \boldsymbol{L_{Dk}}.$$

$\boldsymbol{P_N P_{N-1} \cdots P_{k+1}}$ can be computed accumulatively so than an element of L_D can be inexpensively located in $\boldsymbol{L}$ without explicitly permuting $\boldsymbol{L}$.

Figure 6.2 gives the new row-oriented block sparse forward substitution. In the algorithm, the right-hand side vector b is first permuted by V^{-1}, and the permutation vector V_k is computed accumulately from V. The extra memory requirement for each node is only an $BSIZE \times BSIZE$ submatrix for holding $\boldsymbol{L_{Dkj}}$, and two n-vectors for holding V and V_k. For each node the extra number of integer operations to compute V_k is $O(n \times N)$, which limits the scalability of the algorithm. However on large number of processor we can allocate separate space for V_k on a processor where column block k resides in the processor. In that case the time complexity to compute V_k will be reduced to $O(n)$ but the space complexity will be $O(n \times N/p)$, where p is the number of processors. Under memory constraint the former method should be more effective. On large number of processors when there is enough space to accommodate N/p copies of V_k, the latter one should be better.

(01) V = $\boldsymbol{P_1 P_2 \cdots P_N}$;
(02) $b = V^{-1}b$;
(03) **for** $k = 1$ **to** N
(04) Initialize $buf[1 : bsize(k)]$ to 0;
(05) Initialize $V_k = V$;
(06) **for** $j = 1$ to $k - 1$
(07) $V_k = P_1^{-1} V_k$;
(08) Generate $\boldsymbol{L_{Dkj}}$ from $\boldsymbol{L_{kj}}$ and V_k;
(09) if $\boldsymbol{L_{Dkj}} \neq 0$
(10) $buf[1 : bsize(k)]$ = $buf[1 : bsize(k)] \boldsymbol{L_{Dkj}} \cdot B_j$;
(11) **endfor**
(12) **if** L_{ik} is local
(13) collects the sum of $buf[1 : bsize(k)]$;
(14) B_k = B_k - $buf[1 : bsize(k)]$;
(15) Solve $\boldsymbol{L_{kk}} B_k$ = B_k;
(16) Broadcast B_k;
(17) **endfor**

Figure 6.2: Row oriented block forward substitution to solve $\mathbf{P_1 L_1} \cdots \mathbf{P_N L_N} y = b$. y is stored in b at output. B_i denote $b_{S(k):S(k+1)-1}$, and $bsize(k)$ denote $S(k+1) - S(k)$. In line (08) L_{Dkj} reuses the same space.

In this algorithm at step k, p processors can work on row block k of L_D simultaneously, and different steps can also be concurrent. And it avoids global data redistribution and memory management overhead. However it has some disadvantages. First, it requires some extra integer operations and memory to compute and store V_k. Second, to generate $\boldsymbol{L_{Dkj}}$, one has to search $\boldsymbol{L_k}$, which causes bad data locality. This algorithm is currently being implemented and evaluated.

6.2.2 Back substitution

The parallel back substitution for sparse Gaussian elimination is the same as that of conventional Gaussian elimination. In 1-D data mapping we also take row oriented partitioning, so that at step k p processors can work on row block k simultaneously, and different steps can also be overlapped by using asynchronous communication and taking advantage of elimination tree. The algorithm is outlined in Figure 6.3.

6.3 Parallel triangular solve with 2-D data mapping

6.3.1 Forward substitution

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 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. The row oriented approach can overlap multiple steps easily,

```
for k = N to 1
    Initialize buf_{1:bsize(k)} to 0;
    for j = k+1 to N with U_kj ≠ 0 and U_kj is local
        buf_{1:bsize(k)} = buf_{1:bsize(k)} + U_kj·B_j ;
    endfor
    if L_ik is local
        collects the sum of buf_{1:bsize(k)};
        B_k = B_k - buf_{1:bsize(k)};
        Solve U_kk B_k = B_k ;
    Broadcast B_k ;
endfor
```

Figure 6.3: 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 $S(k+1)-S(k)$.

and each step can be parallelized on more processors because $p_c \geq p_r$ is preferred by sparse LU factorization, but it introduces interprocessor communication and poor data locality to dynamically construct $\boldsymbol{L_{Dkj}}$. Therefore one approach is not always better than the other.

6.3.2 Back substitution

Parallel back substitution is more important than forward substitution in Gaussian elimination because as we will show in the next section, the former one can be performed efficiently during numerical factorization. Back substitution can be done with row oriented approach in the similar way as shown in Figure 6.3, because step k can be done simultaneously by multiple processors and multiple steps can be overlapped. In backward substitution with 2-D data mapping, the major issue is to overlap multiple steps as much as possible, otherwise the speedup is bounded by p_c. The overlapping can come from asynchronous scheduling algorithm, and exploiting the sparsity of the matrix with supernode elimination forest.

6.4 Parallel LU factorization with forward substitution

In Section 6.2 and 6.3, we presented algorithms for sparse forward substitution with 1-D and 2-D data mapping schemes. Regardless of row oriented approach or column oriented approach, the algorithms have to introduce some communication overhead, extra integer operations and memory requirements. We have observed that these overhead can be avoided by solving forward substitution during LU factorization if the right-hand vector(s) is (are) known prior to the factorization.

Our LU factorization algorithm can be easily modified to include forward substitution. It has the following advantages. First, no matter whether in 1-D or 2-D data mapping, the communication overhead of broadcasting right-hand side can be avoided. Second, one need not to explicitly compute V_k permutation matrices and derive $\boldsymbol{L_{Dik}}$. This is especially beneficial for 2-D data mapping when the interprocessor communication are needed to generate L_{Dik}. Third, we can store the right-hand side vector with the last column block, therefore the computation can be done with BLAS-3 subroutines, whereas in forward substitution it can only employ BLAS-2 subroutines.

6.5 Parallel Newton's method for solving nonlinear systems

Newton's method is very important for many scientific and engineering fields like circuit simulation and mechanical engineering. It is an iterative method to solve nonlinear system $F(X) = 0$. In the algorithm, initially an approximate solution X_0 is given. In each iterative step k, it solves a linear system $J(X_{k-1})Y_k = -F(X_{k-1})$, where $J(X)$ is called the *Jacobian matrix* of F, and then set $X_k = X_{k-1} + Y_k$. The method terminates successfully at step i if $||Y_i|| < \epsilon$, where ϵ is a small real number, and the approximate solution to the nonlinear system is X_i; the algorithm aborts when $i > MAX$, where MAX is an integer specified as a parameter.

A noticeable feature of $J(X)$ is that it always has the same structure but different numerical values at each step. Thus our static approach shall apply to the problem very effectively. With static approach, symbolic factorization is done only once, and multiple iterations of numerical factorizations and triangular solves can reuse the same static memory structure, which avoids memory management at runtime which contributes 30% to 40% in SuperLU. Also because the right-hand side vector is always known before numerical factorization, forward substitution can be done efficiently during LU factorization.

We have implemented the idea with 1-D data mapping scheme. The future work is to implement it with 2-D data mapping scheme and evaluate the overall performance.

6.6 Conclusions and future work

In this section, we discussed the issues of parallel triangular solves for sparse Gaussian elimination with partial pivoting. We presented the algorithms for forward substitution and back substitution, which use the same data mapping of sparse LU factorization without global data movement. We proposed a row oriented forward substitution algorithm which exploits parallelism to a great degree and does not introduce any memory management overhead. We have also shown that performing forward substitution during LU factorization could reduce communication and computation overhead, and utilize better computation kernel.

The ideas of the algorithms are being implemented. Our future work is to assess the performance of triangular solves using different data mapping schemes, and implement Newton's method with 2-D data mapping.

Chapter 7

Conclusions and Future Directions

7.1 Summary of contributions

In this thesis, we presented an asynchronous sparse *LU* factorization algorithms using 2-D block mapping, in which we integrated several optimization techniques such as static symbolic factorization, nonsymmetric supernode partitioning, BLAS-3 computation kernel, asynchronous scheduling, and efficient message buffer management. To our best knowledge, this is the first sparse *LU* factorization algorithm with 2-D block mapping, and it is the first time and space efficient algorithm for sparse *LU* on distributed memory machines. This algorithm has achieved breakthrough performance for sparse Gaussian elimination with partial pivoting. Compared to previous performance results on both DMMs and SMMs, the research of sparse *LU* with 2-D data mapping has been very satisfactory and the research of sparse *LU* on DMMs is very promising.

We compared the 2-D algorithm with two different algorithms using 1-D column-block mapping, and showed that 2-D block mapping are more scalable than 1-D block mapping because it requires less space and lower order of communication volume. Therefore 2-D approach has more potential to solve larger sparse matrices and achieve higher performance. However, because 2-D block mapping requires frequent communication, it turns out that 1-D block mapping is more effective when there are small number of processors.

In the thesis, we also proposed a new compression scheme for sparse matrix structure. This scheme has been very successful in improving sequential performance of symbolic factorization for Gaussian elimination and Cholesky factorization. Our experiments show that with the compression scheme the symbolic factorization could be $3.5 - 5.0$ times as fast as the algorithm without compression for a set of small size benchmark matrices, and the space requirement is only $1/5.6 - 1/4$ of the original algorithm. This compression scheme can be easily applied to other sparse matrix computations.

We also designed fast algorithms for supernode partitioning based on the new definition of elimination forest for sparse Gaussian elimination with partial pivoting. Some important properties of the elimination forest are summarized. This definition provides us with some insights of sparse Gaussian elimination with partial pivoting. Based on this definition, we introduced L/U supernode partitioning schemes, and by integrating L and U supernode partitioning, we developed a new amalgamation algorithm which gives a full control over the number of extra nonzeros introduced by amalgamation.

In the thesis, we also formally defined the sparse Gaussian elimination with partial pivoting, and extended it to a block oriented algorithm. Based on this definition, we developed column oriented and row oriented algorithms for sparse forward substitution. The row oriented algorithm could exploit more parallelism than the column oriented algorithm, and does not require data redistribution. We also showed that by performing forward substitution during numerical factorization the overhead introduced by forward substitution can be avoided and better computation kernel can be used. This algorithm is especially useful in Newton's method for solving nonlinear systems.

7.2 Future research directions

7.2.1 Amalgamation algorithm

In Section 3.4 we presented the algorithms for supernode amalgamation without involving permutation, and our experimental studies showed that the algorithms have been very effective in improving performance for many benchmark matrices. However by moving nonconsecutive parent row (column) and child row (column) together, the amalgamation shall generate larger supernodes and deliver better caching performance [**?**].

In [**?**] it has been shown that row or column permutation could undermine the correctness of static symbolic factorization. However if guided by elimination forest, it is possible to preserve the result of symbolic factorization after permutation. For example, the row and column permutation described below will not destroy the result of symbolic factorization:

"If $parent(k) = j > k + 1$, a row and column permutation which moves row k down above row j and moves column k to the right but before column j will preserve the result of symbolic factorization."

Figure 7.1 shows an example of the permutation. In the example, before permutation the average supernode size is 1.25 when $L + U$ supernode amalgamation is performed. However after moving row 2 down above row 4 and moving column 2 right before column 4, the result of the symbolic factorization still holds. And the average supernode size is 2.5 after performing $L + U$ supernode amalgamation. A more effective algorithm for supernode amalgamation could be constructed based on this observation.

7.2.2 Hybrid approach of static and dynamic factorization

Static symbolic factorization has the advantage of static memory structure during numerical factorization and hence it is able to deliver high performance.

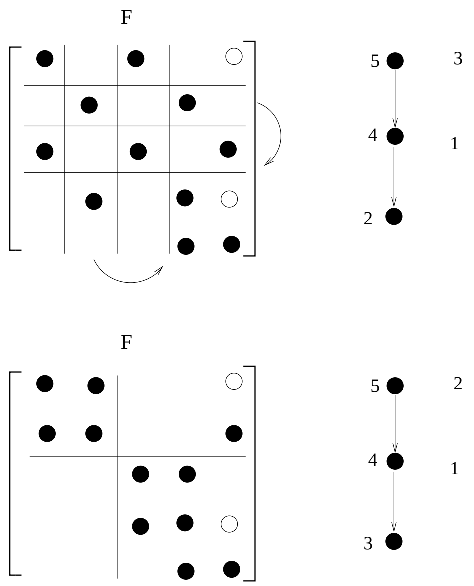

Figure 7.1: An illustration of row and column permutation which preserves the result of symbolic factorization.

However for some ill-conditioned matrices, it could overestimate fill-ins excessively. Figure 7.2 shows a case in which static symbolic factorization generates a dense matrix, but dynamic factorization causes much less fill. On the other hand, dynamic factorization has the advantage of precise manipulation of fill but has high memory management overhead. In SuperLU, dynamic symbolic factorization contributes $30\% - 40\%$ to the total execution time [?]. In addition it is very hard to implement dynamic symbolic factorization on distributed memory machines because supernode structure is unknown until runtime and the data mapping can not be determined before execution.

A hybrid approach of static and dynamic factorization could have the advantages of the both methods. The basic idea is outlined as follows. During analyze phase, the 2-D supernode partitioning is performed based on static symbolic factorization, and the blocks are mapped onto processors based on the partitioning. However the space is not allocated for all the nonzero blocks predicted by the static factorization, instead another round of symbolic factorization is performed.

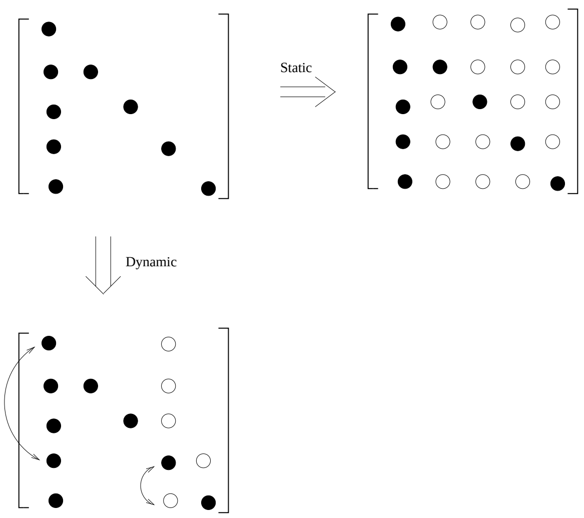

Figure 7.2: An example of excessive overestimation of static symbolic factorization.

The second round of symbolic factorization stops at column d if column d or row d is almost dense which could cause excessive overestimation. Initially the memory space is only allocated for nonzeros predicted by the second round of symbolic factorization, and numerical factorization starts. During numerical factorization, at step d after pivot position for column d is determined, the symbolic factorization continues for the rest of the matrix until it meets another almost dense column or row, and the memory are allocated for the new predicted nonzeros. This process continues until the whole matrix has been decomposed. This approach decreases memory overestimation, and does not introduce too many memory management overhead because there is little chance that a column or a row is almost dense in a sparse matrix, so runtime memory expansion does not happen frequently. This idea is being implemented.

7.2.3 Supernode BLAS

In Section 3.5 we mentioned that in the current implementation of sparse LU factorization, in order to use BLAS-3 computation kernel, we consider an L

block as dense if there is at least one nonzero element in it. This causes significant space overestimation and extra floating-point operations for some matrices, and it obstructs the S^* approach to deliver consistent performance for *all* matrices. The similar problem has also been noticed in other supernode approaches like sparse Cholesky factorization [?] and SuperLU [?], and various efforts have been made to circumvent it.

We have found that the major reason for the problem is not because of the methodology of these approaches, but due to the constraint of the BLAS subroutines. The original purpose of BLAS subroutines was for general dense matrix computations instead of sparse matrix computations, and currently there is no efficient computational kernel designed specifically for sparse computations. It has been a critical issue to design an efficient kernel for sparse codes.

During our research of sparse LU factorization, we have noticed that the L block contains only dense subrows and the U block contains only dense subcolumns. And this is also true for sparse Cholesky factorization and sparse triangular solves. Based on this observation we propose a new kernel "Supernode BLAS" for supernode approaches of sparse matrix computations, which will substantially improve performance and simplify the memory management for sparse matrix computation.

In Supernode BLAS, a source or destination block contains either dense rows or dense columns, and all the other part of the block are zeros. Memory is only allocated for these dense rows or dense columns, and the computations are only performed on these dense structures. Figure 7.3 illustrates an example of matrix-matrix computation of Supernode BLAS. The multiplicand contains only dense rows, the multiplier contains only dense columns, and the result could contain dense rows or dense columns.

A bitmap compression scheme can be used to store the row (column) structure of source and destination blocks. For example for a block containing only dense columns, a bit is assigned for each column. The bit is set to 1 if the column is

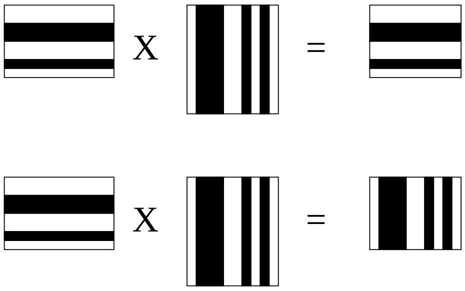

Figure 7.3: An illustration of matrix matrix multiplication in Supernode BLAS-3.

dense, otherwise it is set to 0. Figure 7.4 shows an example of the bitmap for a block which contains only dense columns. With this storage scheme, a nonzero element in the block can be indexed very efficiently, and hence it should deliver competitive performance as dense BLAS subroutines.

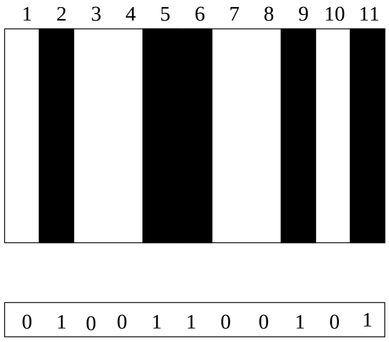

Figure 7.4: An illustration of the bitmap structure for representing a block with dense columns.

Bibliography

[1] C. Ashcraft and R. Grimes. The Influence of Relaxed Supernode Partitions on the Multifrontal Method. *ACM Transactions on Mathematical Software*, 15(4):291–309, 1989.

[2] 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.

[3] Richard L. Burden and J. Douglas Faires. *Numerical Analysis*. PWS Publishing Company, fifth edition, 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. To be published as a book by SIAM.

[7] J. Demmel, S. Eisenstat, J. Gilbert, X. Li, and J. Liu. A Supernodal Approach to Sparse Partial Pivoting. Technical Report CSD-95-883, UC Berkeley, September 1995.

[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. Parallel Sparse LU Factorization with Partial Pivoting on Distributed Memory Architectures. Tech Rep. TRCS97-11, UCSB Computer Science, 1997.

[13] C. Fu and T. Yang. Efficient Run-time Support for Irregular Task Computations with Mixed Granularities. In *Proceedings of IEEE International Parallel Processing Symposium*, pages 823–830, Hawaii, April 1996.

[14] 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.

[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] 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.

[17] A. George and J. W. Liu. *Computer Solution of Large Sparse Positive Definite Systems*. Prince-Hall, Inc, 1981.

[18] 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.

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

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

[21] A. Gerasoulis and T. Yang. On the Granularity and Clustering of Directed Acyclic Task Graphs. *IEEE Transactions on Parallel and Distributed Systems*, 4(6):686–701, June 1993.

[22] J. R Gilbert, C. Moler, and R. Schreider. Sparse Matrices in Matlab: Design and Implementation. *SIAM J. Matrix Analysis and Applications*, 13:333–356, 1992.

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

[24] A. Gupta and V. Kumar. Highly Scalable Parallel Algorithms for Sparse Matrix Factorization. Technical Report TR 94-63, Computer Science, Univ. of Minnesota, August 1994.

[25] 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.

[26] M. Heath, E. Ng, and B. Peyton. Parallel Algorithms for Sparse Linear Systems. *SIAM Review*, 33(3):420–460, September 1991.

[27] J. W. H. Liu. Computational Models and Task Scheduling for Parallel Sparse Cholesky Factorization. *Parallel Computing*, 18:327–342, 1986.

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

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

[30] S. L. Scott. Synchronization and Communication in the T3E Multiprocess. In *ASPLOS-VII*, October 1996.

[31] 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.

[32] A. H. Sherman. On the Efficient Solution of Sparse Systems of Linear and Nonlinear Equations. Research Rpt. 46, Dept. of Computer Science, Yale University, New Haven, CT, 1975.

[33] T. Stricker, J. Stichnoth, D. O'Hallaron, S. Hinrichs, and T. Gross. Decoupling Synchronization and Data Transfer in Message Passing Systems of Parallel Computers. In *Proc. of International Conference on Supercomputing*, pages 1–10, Barcelona, July 1995.

[34] C. Sun. Parallel Sparse Orthogonal Factorization on Distributed-Memory Multiprocessors. *SIAM Journal on Scientific Computing*, 17(3):666–685, May 1996.

[35] T. Yang and A. Gerasoulis. PYRROS: Static Task Scheduling and Code Generation for Message-Passing Multiprocessors. In *Proc. of 6th ACM International Conference on Supercomputing*, pages 428–437, 1992.

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